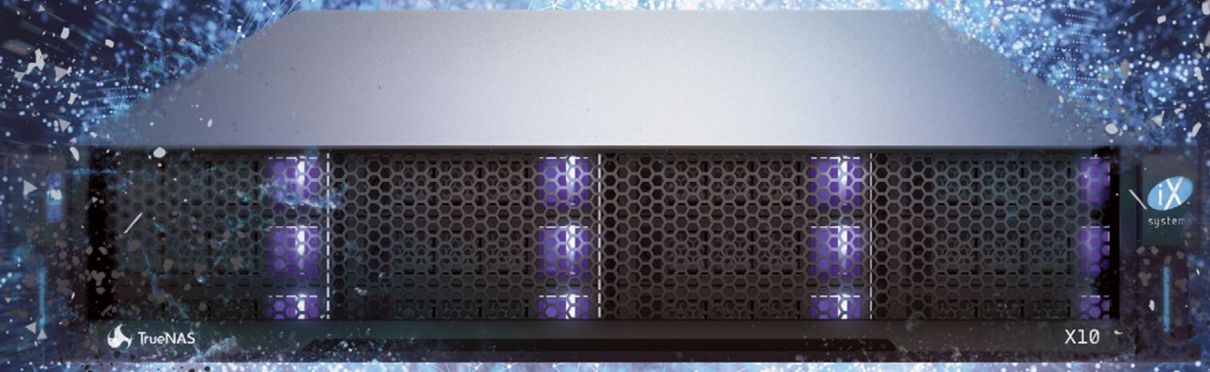
FreeBSD™ JOURNAL

May/June 2017

Security

- FreeBSD's Firewall Feast
- GamingBSD
- Toward Oblivious Sandboxing with Capsicum
- Cadets: Blending Tracing and Security
- Whats Going On with CloudABI?
- pfSense & Security

BORN TO DISRUPT



MODERN. UNIFIED. ENTERPRISE-READY.

INTRODUCING THE TRUENAS® X10, THE MOST COST-EFFECTIVE ENTERPRISE STORAGE ARRAY ON THE MARKET.

Perfectly suited for core-edge configurations and enterprise workloads such as backups, replication, and file sharing.

- ★ **Modern:** Not based on 5-10 year old technology (yes that means you legacy storage vendors)
- ★ **Unified:** Simultaneous SAN/NAS protocols that support multiple block and file workloads
- ★ **Dense:** Up to 120 TB in 2U and 360 TB in 6U
- ★ **Safe:** High Availability option ensures business continuity and avoids downtime
- ★ **Reliable:** Uses OpenZFS to keep data safe
- ★ **Trusted:** Based on FreeNAS, the world's #1 Open Source SDS
- ★ **Enterprise:** 20TB of enterprise-class storage including unlimited instant snapshots and advanced storage optimization for under $10,000

The new TrueNAS X10 marks the birth of a new entry class of enterprise storage. Get the full details at iXsystems.com/TrueNAS.

iX systems™

Table of Contents

May/June 2017

SECURITY

COLUMNS & DEPARTMENTS

3 Foundation Letter
Shh, don't tell anyone.
By George Neville-Neil

42 New Faces
In this installment, the spotlight is on Eugene Grosbein, a longtime contributor who transitioned to a ports committer in March 2017.
By Dru Lavigne

44 Book Review
Designing BSD Rootkits: An Introduction to Kernel Hacking by Joseph Kong. *Reviewed by Steven Kreuzer*

46 svn Update
This installment highlights recent improvements designed to keep you and your data safe.
By Steven Kreuzer

48 Events Calendar
By Dru Lavigne

FreeBSD's Firewall Feast
FreeBSD is famous for all sorts of fantastic features. It's somewhat infamous, however, for having three different firewalls. *By Michael W Lucas*

4

pfSense® and Security
pfSense® software provides users with the main features offered by most commercial firewalls, but with the primary advantages of costing less and being open source.
By James Dekker, Renato Botelho, Luiz Otavio O. Souza

6

CADETS: Blending Tracing & Security on FreeBSD As part of a research project, CADETS is adding new security primitives to FreeBSD as well as leveraging existing primitives to produce an operating system with maximum transparency. *By Jonathan Anderson, George V. Neville-Neil, Arun Thomas, Robert N. M. Watson*

12

Toward Oblivious Sandboxing with Capsicum
Capsicum, a principled and coherent design for software compartmentalization, has taken recent strides toward a new security model. *By Jonathan Anderson, Stanley Godfrey, Robert N. M. Watson*

18

Gaming BSD Please be in your most paranoid state when reading this article—ready to wear the two hats of security, the black one and the white one.
By Roberto Fernández

26

What's Going On with CloudABI? As CloudABI is part of FreeBSD 11 and most new features have already been merged into 11-STABLE, CloudABI has become an easy-to-use tool for creating secure and testable software. *By Ed Schouten*

32

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher •	Walter Andrzejewski walter@freebsdjournal.com
Editor-at-Large •	James Maurer jmaurer@freebsdjournal.com
Copy Editor •	Annaliese Jakimides
Art Director •	Dianne M. Kischitz dianne@freebsdjournal.com
Office Administrator •	Michael Davis davism@freebsdjournal.com
Advertising Sales •	Walter Andrzejewski walter@freebsdjournal.com Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
5757 Central Ave., Suite 201, Boulder, CO 80301
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org
Copyright © 2017 by FreeBSD Foundation. All rights reserved.

This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER from the Board

Shh, don't tell anyone.

It's a secret. We've compiled an entire issue on "security"! Well, actually, please do not keep this issue a secret as it contains an excellent roundup of interesting technologies that help make FreeBSD more secure as well as ways in which FreeBSD itself is applied to the problems of secure systems.

We kick off the issue with Michael Lucas's piece on the Firewalls of FreeBSD. FreeBSD contains the PF, IPFilter, and IPFW firewalls, systems that ostensibly implement the same features, but have several interesting differences. Next is an article on the CADETS research project. With CADETS, we're blending tracing—via DTrace—with the audit system and other components of FreeBSD to produce a platform based on the idea that having transparency—i.e., who does what to whom—on a computer allows us to build a more secure system overall. Then Jonathan Anderson, Stanley Godfrey, and Robert N. M. Watson take us on a journey through the sandbox, via their article on Capsicum, the native, capability-based isolation technology that is present in FreeBSD. Capsicum resurrected the concept of Capability Systems, first researched in the 1970s. Capabilities were once considered too computationally expensive for practical systems, but the work done by Robert N. M. Watson in his PhD thesis showed that a capability-based system could be built for Unix-like systems that was both practical and efficient. James Dekker, Renato Botelho, and Luiz Otavio O. Souza guide us through the main security features of pfSense in their article on this popular, FreeBSD-derived, open-source firewall product. FreeBSD can be found in many interesting environments, including devices that implement games of chance. In Europe, you'll see these games in corner bars and other venues. Roberto Fernández tries to induce a bit more paranoia than is typical by explaining what it takes to secure a gaming device with FreeBSD. We round out this issue with an update on CloudABI, by Ed Schouten. A tool that's part of FreeBSD, CloudABI helps create secure software that is easier to test and verify.

Steven Kreuzer also highlights the security theme, reviewing Joseph Kong's book *Designing BSD Rootkits: An Introduction to Kernel Hacking*, as well as covering all that's new in the FreeBSD source tree in his svn update column. Through Dru Lavigne's New Faces, we're introduced to Eugene Grosbein, a longtime contributor who transitioned to a ports committer in March. And don't forget to check out the BSD-related events we should consider putting on our calendars.

It's time now to sit back, put on your tinfoil hat, and enjoy reading this latest issue of the *FreeBSD Journal*.

George Neville-Neil
Editor in Chief of the *FreeBSD Journal*
Director of the FreeBSD Foundation

FreeBSD's Firewall Feast

By Michael W Lucas

FreeBSD is famous for all sorts of fantastic features, such as ZFS, jails, bhyve virtualization, and the Ports Collection. It's somewhat infamous, however, for having three different firewalls: PF, IPFilter, and IPFW. Where did all these firewalls come from, and why are they all still in the system?

The IT industry has repeatedly abused, stretched, and tormented the word firewall to fit all sorts of different products. When someone uses firewall, ask them exactly what they're talking about. Do they mean a caching HTTP proxy like Squid or Varnish? A generic proxy like relayd? Or a TCP/IP packet filter?

All of FreeBSD's firewalls are packet filters. They control which TCP/IP addresses and ports can connect to the host. If your FreeBSD host passes packets between interfaces, then the firewall controls which traffic gets forwarded, which gets silently dropped, and which is sent back to the source with a letter of complaint. A packet-forwarding packet filter is the original firewall.

The firewalls all have a common core feature set considered the minimum for a modern packet filter. They can track the state of a TCP/IP connection and permit traffic based on existing connections. They can all return resets or silently drop connections. All can manage non-routable addresses and perform network address translation. They all work with lists of rules defining how to respond to traffic from different IP addresses and network ports. Incoming packets are compared to the list of rules until they are permitted or rejected.

The firewalls have their own unique features, however. How do you choose between them?

IPFW is the oldest FreeBSD packet filter, dating from 1995. IPFW maintains its rules in a numbered list. You add a rule to an existing ruleset by giving it a rule number. Rule numbers range from 1 to 65534. IPFW rules are very dynamic, and can be altered programmatically. Numbered rules do impose some limits, however. Every time I design an IPFW ruleset, I eventually wind up having to renumber the rules because, somehow, I need to cram too many rules in between two other rules. This is a limitation of my imagination when creating my ruleset, though, not any limitation in IPFW.

The most interesting unique feature in IPFW is dummynet, which lets you selectively degrade traffic. Why would you want to make a connection worse? Well, for one thing, it's kind of fun to simulate an ADSL link to the moon. But the ability to impose excess latency and bandwidth limitations on a connection can save a global organization days or weeks of time and thousands of dollars in developer travel expenses. More than once, I've set up an IPFW box to simulate a connection from the other side of the world, or with multiple profiles to simulate several different countries. A user in South Korea with a gigabit Internet line will have a very different experience than a user who's closer but has less bandwidth. Dummynet lets you simulate their experience locally.

IPFW also has the highest performance of any FreeBSD firewall, although that only becomes apparent at tens of gigabits per second.

IPFilter, or IPF, is a platform-independent firewall, and came to FreeBSD in the late 1990s. You can use IPFilter on FreeBSD, Solaris, SunOS, HP-UX, NetBSD, OpenBSD, and Linux. It has a configuration syntax similar to PF. If you must run one firewall across multiple operating systems, IPFilter is for you.

IPFilter is not in active development, but it can block, permit, and translate packets. It's largely feature-complete.

PF is the newest FreeBSD firewall, and is quite popular among younger sysadmins. It originated with the OpenBSD project, as their replacement for IPFilter. It's quite popular among younger sysadmins who didn't grow up with IPFW.

PF is perhaps the most popular FreeBSD firewall. PF has a simpler configuration syntax than IPFW. While you can alter the ruleset programmatically, you must configure the ruleset to permit such alterations. While mailing list archives discuss occasional problems, such as handling IPv6 fragmentation, those issues were solved years ago.

While FreeBSD's PF originated with OpenBSD, that import happened several years ago. FreeBSD's PF has diverged from the original import, and OpenBSD's PF has continued its natural evolution. Chances are that the two will continue to diverge.

All this information is nice, but which should you choose?

When you're studying PF configuration, be sure that you use documentation relevant for FreeBSD, not current OpenBSD documentation. The chances of FreeBSD taking a new import of OpenBSD's PF are very slim. PF is also incompatible with the vimage virtual networking stack used by jails.

All this information is nice, but which should you choose? That depends on your environment. Most sysadmins with small servers will find PF's simplicity a win.

If you want one firewall software across multiple Unix-like operating systems, use IPFilter.

If you need to pass tens of gigabits per second, simulate a bad connection, have unlimited ability to programmatically alter rules, or require advanced jail networking, use IPFW.

If you already have one deployed, of course, keep using it. They all work fine for the essential firewall task of filtering packets. Combined with an application proxy or cache such as Squid, relayd, or varnish, FreeBSD can go head-to-head with any commercial firewall vendor and win. ●

MICHAEL W LUCAS is the author of several books on FreeBSD, including *Absolute FreeBSD* and the *FreeBSD Mastery series.* Learn more at www.michaelwlucas.com.

SEE TEXT ONLY

FreeBSD is an operating system well-known for its advanced network stack and security. It provides thousands of third-party software packages through the ports collection, making it possible for expert users to have a great firewall running with a minimum of investment. Other users with less expertise or even no experience with FreeBSD, and in some cases more money, prefer to run their security solutions using commercial products.

pfsense® and Security

By James Dekker, Luiz Otavio O. Souza, and Renato Botelho

pfSense® software exists in the middle of these two worlds. pfSense is an open-source firewall/router distribution running on top of FreeBSD. All configuration occurs via an easy-to-use web GUI, and many functions are provided using other open-source applications (e.g., Unbound, Strongswan, OpenVPN, etc.). pfSense software provides users with the main features offered by most commercial firewalls, but with the primary advantages of costing less and being open source. Open-source software can be audited and fixed when its behavior or security is in doubt. The open-source model also allows anyone to fix broken code, while closed source can only be fixed by the vendor.

pfSense software was born in 2004, as a fork of m0n0wall. Since then it has been continuously developed and improved, making it a trusted product used by hundreds of thousands of users around the world. Currently there are over 550,000 active installations of pfSense software.

Netgate is the company behind pfSense and employs core developers, making it possible to have people working on and supporting it every day of the year. Netgate offers 24/7 support and other professional services to help users design their network from scratch, review existing configurations, and migrate environments from most known commercial products to pfSense software to meet the customer's requirements.

Improvements in New Versions

After a long period of being based on FreeBSD 8, pfSense software version 2.2, based on FreeBSD 10.3, was released January 23, 2015. We have continued tracking FreeBSD closely, and as a result, the 2.3 version, released April 12, 2016, was based on FreeBSD 10.3, (released March 28, 2016). pfSense software 2.3 brought a large number of new features and improvements. The most significant changes were a complete rewrite of webGUI using Bootstrap and the underlying system, including the base system and kernel, being converted entirely to FreeBSD pkg.

Having pfSense software use pkg brought three large benefits. First, pfSense software updates were historically distributed as a monolithic image that was extracted on top of the currently installed version. There were no records of any system files. Using FreeBSD pkg made it possible to distribute minor upgrades easily and made the process of testing and validating a new upgrade much faster.

Second, using pkg to upgrade every piece of pfSense software makes it possible to detect whether or not the kernel is being upgraded. With version 2.4, this information is used by pfSense-upgrade to decide if the system needs to be rebooted or not. If the answer is no, it will use reroot as an alternative and the system will restart all services and be back online faster using the same running kernel.

Finally, starting with version 2.1, pfSense used PBIs to distribute additional packages. This was a good choice at the time because each PBI contained all required libraries and other binary dependencies so as to not pollute the pfSense core with additional files. With pkg, all primary and additional packages are built together in the same environment using poudriere. Poudriere takes care that dependencies are tracked correctly and that all unnecessary files are removed when a package is uninstalled, leaving the system clean.

The conversion of the WebGUI to Bootstrap made the user interface more modern, easier, and intuitive, while leaving the code both more readable and easier to extend, resulting in a renewed level of contributions to the project. The number of pull requests submitted to main pfSense repository at github more than doubled from 384 to 955 during 2.3 webGUI conversion time.

We continue to track FreeBSD source closely. Our next major release, pfSense software version 2.4, which is in BETA as we write this article, will be based on FreeBSD 11. With this version, 32-bit architecture and the nanobsd model were discontinued, but we now offer images for ARM platform, more specifically to Netgate SG-1000 (uFW). The old bsdinstaller has been replaced by bsdinstall, the same installer used by FreeBSD, which brings important features such as UEFI and ZFS support.

FreeBSD 11 also brings an updated 802.11 stack, which will bring pfSense software numerous improvements when used as a WiFi client or access point. Appliances offered in the Netgate/pfSense store with native wireless are fully compatible with FreeBSD 11 and are ideal for applications including research and penetration testing.

During 2.3 and 2.4 development, we drastically reduced the number of patches we carried against the FreeBSD source code to accommodate features required by pfSense users. Changes that were of interest to upstream FreeBSD users were contributed to FreeBSD, while other patches were removed and pfSense was reworked to have the same result without patching FreeBSD. As a result of this work, pfSense software version 2.4 is currently running on top of a FreeBSD tree that is much closer to stock. This yields the advantage that we can now move to a new FreeBSD major release more quickly than ever before. In the past, and under different leadership, the project took two years to move from FreeBSD 8.1 to FreeBSD 8.3 and another 20 months to move from FreeBSD 8.3 to FreeBSD 10.1.

Packages

One of the major benefits of pfSense is the ability to install additional packages that provide extra functionality such as Snort, Suricata, and Squid. In the past, the number of available additional packages became unworkable as the maintainers of some of them lost interest, leaving some packages to stagnate.

During the pfSense 2.3 release process, while we migrated from the old pfSense-packages repository to a FreeBSD ports-based repository, we decided to clean up and remove packages that have been deprecated upstream, no longer have an active maintainer, or were never stable.

Remaining packages were converted to the new Bootstrap model and reworked to become available as a FreeBSD pkg. Following the release of version 2.3, the community came together to migrate and maintain some packages that had been removed. After review, some of these packages were accepted and reintroduced to the list of available packages.

Captive Portal

The Captive Portal function in pfSense software allows securing a network by requiring a username and password (or only a click-through entered on a portal page). If authentication is used, this can be performed with native user management within pfSense or an external authentication server such as an Active Directory, LDAP or RADIUS server.

The pfSense firewall is implemented principally by pf, one of the three packet filter implementations available in FreeBSD. When Captive Portal is enabled, it uses ipfw, another packet filter implementation available in FreeBSD. Until pfSense software 2.3, we had a heavily modified version of ipfw to make it possible for Captive Portal to work with multiple instances. During the move to FreeBSD 11, due to a huge rewrite of ipfw on FreeBSD, we decided it was a good time to remove the big ipfw patch and make Captive Portal work with native ipfw, or at least with just a few modifications.

All Captive Portal pieces that interact with ipfw were reworked, and, as a result, we were able to eliminate one big extra patch that led us closer than stock FreeBSD. Necessary changes done during this work will be reviewed and upstreamed as soon as 2.4 is released.

Security

pfSense software's main goal is to be a security appliance, and because of that, we take it seriously. In 2014, we announced Security Advisories containing all security-related issues that are discovered, along with their corresponding CVE IDs.

A significant number of security flaws related to the webGUI are caused by the use of HTTP GET. To mitigate this attack, vector pfSense software 2.4 has converted many parts of the webGUI to work only via HTTP POST.

Code Review

In 2016, Netgate enlisted the help of Infosec Global to conduct a top-to-bottom, post-commit audit of pfSense software version 2.3.2. Conducting an independent code review helps improve the quality of the product.

For this project, we provided Infosec Global with the Netgate XG-2758 1U Security Gateway Appliance with pfSense software 2.3.2 installed with a default production configuration.

Infosec Global scores threats on a bottom-up percentage scale, with 0% being a perfect score and 100% being most critical. As indicated in the audit report, pfSense 2.3.2 scored an outstanding 1%, which included concerns that were already mitigated during the timeframe of the audit process with the release of pfSense software version 2.3.2_p1. Other issues raised do not apply to the firmware version reviewed. The complete report is available for review at https://www.netgate.com/assets/ISG_Netgate_2016.pdf

OpenVPN

With the release of pfSense software version 2.4, OpenVPN has been upgraded to version 2.4.0. That is a significant upgrade that includes support for a number of new features, including support for AEAD ciphers such as AES-GCM. We also added support for Negotiable Crypto Parameters (NCP) to control automatic cipher selection between clients and servers.

Let's Encrypt

In February 2017, a new package called ACME was made available for releases greater than 2.3.2 in Package Manager. This package interfaces with the Let's Encrypt project to handle the certificate generation, validation, and renewal processes.

Let's Encrypt is an open, free, and completely automated Certificate Authority from the non-profit Internet Security Research Group (ISRG). The goal of Let's Encrypt is to encrypt the web by removing the cost barrier and some of the technical barriers that discourage server administrators and organizations from obtaining certificates for use on Internet servers, primarily web servers. Most browsers trust certificates from Let's Encrypt. These certificates can be used for web servers (HTTPS), SMTP servers, IMAP/POP3 servers, and other similar roles that utilize the same type of certificates.

Certificates from Let's Encrypt are domain validated, and this validation ensures that the system requesting the certificate has authority over the server in question. This validation can be performed in a number of ways, such as by proving ownership of the domain's DNS records or hosting a file on a web server for the domain.

By using a certificate from Let's Encrypt for a web server, including the webGUI in pfSense, the browser will trust the certificate and show a green check mark, padlock, or similar indication. The connection will be encrypted without the need for manually trusting an invalid certificate.

Translations

While pfSense has always been a project of contributions from the community, efforts to translate pfSense started many years ago, but were never 100% complete for any language. We were also never able to provide a central place or good toolchain for contributors of translations for pfSense. For pfSense software version 2.4, we decided to bring this subject back and added pfSense software to the Zanata translation website. Once we had announced this setup, it was a short time until we received a full translation into Spanish, Russian, Norwegian, Chinese (Simplified, China), Chinese (Taiwan) and Bosnian, with incredible progress in German and many others. If you want to see pfSense software translated to your native language, join us!

Contributing to Upstream

Our main upstream is FreeBSD. Both FreeBSD source and ports repositories are the bases used to build and run pfSense software. During the past two years, Netgate engineers have contributed 155 commits to the FreeBSD src repository and 35 commits against the FreeBSD ports collection.

But FreeBSD is not our only upstream; we also run other software provided by third-party projects such as Strongswan, OpenVPN, and dpinger. pfSense developers consistently strive to submit fixes for upstream projects every time a bug is found or a new feature is implemented. This is part of an ongoing effort to improve the software for the entire user base.

pfSense Software in the Cloud

With today's cloud platforms, you can easily extend an on-premise IT environment into the cloud in a manner similar to setting up and connecting to a remote branch office. While these services provide excellent tools for connecting VPNs and setting up access lists, they do not provide full visibility and manageability into the endpoints, nor do they secure the connectivity of critical cloud services.

The Netgate-provided images for pfSense software on AWS and Azure deliver advanced routing, firewall, and VPN functionality for your public cloud infrastructure at a lower cost than other solutions. The pre-built pfSense AWS and Azure images have features similar to both the pfSense hardware appliances and the VMware Certified pfSense available from Netgate.

ARM Support on pfSense 2.4

We had just released pfSense 2.3 when the prototypes of our first ARM system landed on our desks. It was then we knew that the ARM support on FreeBSD had evolved so quickly between 10 and 11, to the point where it was extremely difficult to port many of the new features and fixes. It became clear that FreeBSD 11 would be the only reasonable option and we knew that moving on was the best way to provide the optimal ARM experience.

Since then, Netgate has upstreamed all the code developed, tested, and used daily on the SG-1000. A few contributions specific to the SoC used in this platform include:

- numerous platform and Ethernet fixes to increase system stability and performance
- dual Ethernet support
- management of the integrated Ethernet switch

The introduction of a new, less forgiving ARM architecture to our development environment also helped us spot a few misbehaving sections of code that would have passed unnoticed on a more forgiving architecture. The outcome of fixing these is code correctness and better overall portability.

Today, the ARM support in pfSense provides a seamless experience for any user of this newly introduced architecture. Everything was carefully worked out to provide a reliable, smooth, and incredibly intuitive experience, to the point that you will likely forget you are using a tiny ARM device (unless you manage to remember where this little device lives—or hides—in your office).

Thanks to the pfSense pkg support for updates, the updates on the SG-1000 are bliss and the whole system is updated from the WebGUI (FreeBSD kernel and base, pfSense base and packages), which makes the management of these devices terrific.

The SG-1000 has significantly increased the number of FreeBSD ARM devices in the world. As of this writing, we are actively working on our next ARM system, a dual core ARM with more speed and more hardware features.

Since pfSense began as a fork of m0n0wall, it has continued to evolve into a product that drives the wedge deeper and deeper into the enterprise market space, while still meeting and exceeding the needs of the SMB, prosumer, and home user. What once was a project to provide an extensible and easy-to-use software product for an embedded hardware system has grown into a product that will now run on hardware from embedded ARM devices smaller than a credit card to high availability systems that can take up 2U of rackspace and even larger clustered configurations. This highly versatile, increasingly stable, and secure software product would not be possible without the massive support and involvement of both the community and the resources of Netgate. It's only with the combination of these that we are able to continue pushing the envelope in terms of what is possible for not only FreeBSD, but also, as time will tell, a return to our roots in providing an extensible, high-throughput, easy-to-use software product for embedded devices. ●

RENATO BOTELHO started using FreeBSD on version 3.2 after a friend gave him an installation CD and said, "Install it and never look back." His first contributions to FreeBSD ports tree in 2004 led to his becoming a committer the next year. Likewise, his involvement with pfSense in 2009 was soon followed by his employment at Netgate. He lives in a small city in Brazil with his wife, son, daughter, and two dogs. When he is not in front of a computer, he enjoys running, CrossFit, beer, and barbecues—not necessarily in that order.

JAMES DEKKER is a diehard fan of pfSense and a Production Support Analyst for Netgate. He enjoys securing systems and the networks that connect them. He has been working with *nix systems for some time, primarily focused on security, networking and systems administration. He has been using pfSense since version 2.0 and providing support to the community since 2.1. When James is not working or breaking things in his office, he can be found on the beach, fishing, or working on the family ranch. He lives on the Treasure Coast in Florida with his wife, dog, and cat.

LUIZ OTAVIO O. SOUZA is a software engineer at Netgate and a FreeBSD committer.

SEE TEXT ONLY

CADETS

Blending Tracing & Security on FreeBSD

The FreeBSD operating system has been used in many products that require strong security properties—from storage appliances, to network switches and routers as well as gaming consoles. Over the 20-plus years of FreeBSD's development, there have been significant additions to the system in the area of security.

By Jonathan Anderson, George V. Neville-Neil, Arun Thomas, and Robert N. M. Watson

The Jails [1] system introduced what today would be considered a lightweight version of virtualization. The Mandatory Access Control (MAC) framework was added in 2002 to provide fine-grained, system-wide control over what users and programs could do within the system [2]. The Audit subsystem added the ability to track—on a system-call-by-system-call basis—what actions were occurring on the system and who or what was initiating those actions [3]. More recently Capsicum has introduced capabilities into the operating system that are the basis for application sandboxes [4].

As part of a new research project, the Causal Adaptive Distributed, and Efficient Tracing System (CADETS), we are adding new security primitives to FreeBSD as well as leveraging existing primitives to produce an operating system with the maximum amount of transparency. Apart from the security implications, having better visibility into systems will help us to improve overall performance and provide new runtime debugging tools.

In this article, we'll cover our work with DTrace and the Audit system, which form the core components of the work, and talk about the challenges of using DTrace as an **always on** tracing system to track security and other events.

Starting at the Beginning

One of the main components of FreeBSD we are exploiting in CADETS is DTrace. Originally developed for Sun's Solaris operating system in the early years of the 21st century [5], DTrace was meant to solve the following problem: most software systems have some sort of logging framework, which usually depends on the ubiquitous `printf` function and formats text and sends it to some form of console.

```
printf("Hello world!");
```

All C programmers know the code shown above and every programmer knows the equivalent idiom in their own language, whether it's Python, Rust, Go, or PHP. There are a few problems with using print statements for logging systems. The first problem is that print statements have a high performance overhead. If you've ever wondered about how much work is done on the programmer's behalf by `printf` then see Brooks Davis's "Everything you ever wanted to know about 'hello, world.'" [6] Using print statements for a logging system in code that is supposed to have, otherwise, low overhead, is not an option, and so most logging systems are enabled or disabled at either compile time, using an `#ifdef/#endif` statement, or at runtime, by wrapping the logging in an `if` statement. An example of this idiom in the C language is shown below.

```
#ifdef LOGGING
if (log)
        printf("You have written %d bytes", len);
#endif /* LOGGING */
```

The next problem with print-based logging systems is that they are both static and prone to errors. If the programmer did not expose a piece of information via the logging system before the code was built, then it will not be possible, without modifying the code, to learn about any other data upon which the same function might be operating.

Together, these two problems mean that most high-performance systems, such as operating systems, are shipped without logging enabled, and when logging is enabled, the data that is available is limited to whatever the original programmer wished to expose.

As open-source developers, we are used to the idea that we can "just recompile the code," but in production systems, that's not always possible. Imagine you have sold a system to a large bank. At 3 a.m., the system has a fault of some sort and logs an error. Someone in the IT department gets a message reporting the fault, they call support, support calls a programmer, the programmer then says, "Stop the system, rebuild the code, and rerun it with logging turned on." Handling errors in that way is terrible both for the customer and for the developer. The customer will be annoyed at having to rebuild and restart their system, and the developer is unlikely to get helpful information because the error is now far in the past. Enter Dynamic Tracing.

DTrace was designed to always be available for use, without reducing system performance and without running the risk of outright crashing the system. The clearest way to think about DTrace is as a runtime debugger with significant logging and statistical capabilities. DTrace avoids the overhead of `printf` based logging systems by using a few tricks to subvert the execution of compiled code. The complete details are covered in *The Design and Implementation of the FreeBSD Operating System* [7], but, briefly, DTrace can override the entry or exit point of any function that is compiled into the kernel or into a user-space program. The way in which functions are collected into libraries and programs is a well-defined process, and each function has a set of instructions that indicate where the function begins. When DTrace traces a function, it replaces a few instructions with some of its own so that when the code reaches that point, DTrace's code gets called first, and it can collect and process the function's argument. The practical upshot of this is that when DTrace is not in use, it has zero overhead.

Tracing for Security

How can we apply DTrace to the problems of security? One aspect of security is finding where the bad actor lies in a system. A key question to ask when looking at a system is "Who did what to whom and when?" We'll refer to this as "Question 1." Establishing the tree of operations such that we can trace it back to its root is one part of forensic analysis that can help us find our bad actor and also show us what we need to change to prevent future security breaches.

Imagine that, regardless of the performance cost, we had complete transparency into every operation performed on a computer system. With sufficient time, analysis, and tooling, we would be able to take the output generated by the tracing system and find out the answer to Question 1.

DTrace gives us the basis on which to build such a system, but there remain some challenges. In the previous section, we stated that using some clever tricks to replace certain instructions at runtime, DTrace would have zero overhead **when not in use**. That feature of DTrace was what made it viable to ship it, by default, with Solaris, and the FreeBSD and Mac OS in the first place. It was not until there was a way to ship a tracing system that didn't have a performance impact on a running system that such a system could be fielded. What happens when DTrace begins tracing? Depending on what is being traced and how much data is being collected, the overall performance of the system will be impacted to a greater or lesser degree. If the overhead introduced by tracing gets too high, then the kernel will terminate the tracing as a form of self-preservation. The second tenet of the DTrace system, that the tracing system must not unduly tax the system, is one of the key challenges of using DTrace as a security technology. An attacker that knows there might be tracing will first cause there to be a great deal of irrelevant load on the system, causing the tracing system to exit, and then they will go about attacking the system. Another component of the CADETS project looks at the provenance of traces in order to thwart such attacks on the tracing system itself.

The majority of the current use cases for DTrace involve using it as a runtime debugger, turning on tracing when someone suspects there is a problem on a system, rather than leaving the tracing running all the time. A small subset of users have built complex telemetry systems around DTrace, including Fishworks [8], but in these first attempts at **always-on tracing** the number of things being traced was a small subset of the possible tracepoints. To build a system that has complete visibility, we need to not only have always-on tracing, but to increase the performance of the tracing system such that the overhead collecting the data does not overwhelm the system's ability to do productive work. Another requirement of a tracing system targeted at security is that trace records cannot be dropped or lost due to high load. DTrace was designed in such a way that under high load it was acceptable to drop trace records, long before the kernel might terminate the `dtrace` collection process due to the system being unresponsive. A system that is trying to collect trace data for later forensic analysis turns that concept on its head.

Any system where tracing is always on will generate a lot of data, and that data will need to be analyzed to track down attackers and how they are able to compromise the system. There are several systems for taking arbitrary textual output from various tools and trying to make some sense of it, including Splunk [9]. The goal of the CADETS project is to produce data for use by such tools. It will be much easier for consumers if the data is in a machine-readable format.

Trace Records for Software Tools

As we began our work using DTrace as an

always-on tracing system, we quickly realized that parsing the voluminous output generated by the system would present a problem not only for human analysts, but also for any tools that would be consuming the data. One of the first features we added to DTrace for CADETS was **machine-readable output**. Using the `libxo` library, we converted DTrace to not only produce plain text, but also XML, JSON, and HTML, the three output formats supported by `libxo`. At the time that we added machine-readable output, the Illumos version of DTrace did have a way of outputting JSON, but this was via a print-like operation rather than a pervasive change. In the CADETS version of DTrace, a single command-line option changes all the output the user sees from DTrace from plain text into a machine-readable format.

~~~
```
# dtrace -n 'syscall::write:entry'
dtrace: description 'syscall::write:entry' matched 2 probes
CPU     ID                    FUNCTION:NAME
     0   59780                  write:entry
     0   59780                  write:entry
Example 1```

```
# dtrace -O json -n 'syscall::write:entry'
dtrace: description 'syscall::write:entry' matched 2 probes
CPU     ID                    FUNCTION:NAME
{
     "probe": {
        "timestamp": 3594774042481656,
        "cpu": 1,
        "id": 59780,
        "func": "write",
        "name": "entry"
 }
Example 2
~~~

The code samples above show the differing output between the plain, textual output, and machine-readable output. Our example asks DTrace to trace all calls into the `write` system call. Example 1 shows the default, textual output from the `dtrace` command, which is arranged in columns. To write a tool that parses the output requires knowing quite a bit about the output, because the columns are only labeled at the start of the output. Example 2 shows the same tracepoint, but with the addition of the `-O json` command line argument, instructing `dtrace` to give us all output in JSON format. The machine-readable output tags each element, starting with the `probe`, which is an object that contains several elements, including the `cpu`, `id`, `func` and `name`, all of which appeared, untagged, in the machine-unreadable output of Example 1. One addition for machine-readable output is the timestamp element, which reports the time that the probe fired, in nanoseconds, since the UNIX epoch. While a DTrace script can output the time using either the `timestamp` or `walltime stamp` variables, we believed that having a `time stamp` in every machine-readable record would simplify building tools for security forensics. While probes may fire in parallel on different cores, indicated by the `cpu` variable, modern Intel-based systems have a synchronized timestamp, which DTrace uses, meaning that the time stamps are an excellent indication of the order of operations on a single system.

DTrace and Audit

The `audit` subsystem has been a part of FreeBSD since 2004 and is an optional kernel component, along with a user space daemon `auditd`, which implements a "fine-grained, configurable logging of security related events [10]. It was built to meet the "Common Criteria (CC) Common Access Protection Profile (CAPP) evaluation," a security standard set out by the U.S. Government [11]. The audit subsystem adds a set of handcrafted tracepoints via C macros to parts of the kernel where access to data and resources take place. For example, in a system with `audit` enabled, any time a file descriptor is accessed, a note is made in an audit record. Audit records are periodically flushed to permanent storage.

Our recent work with DTrace and security led us to desire a bridge between the `audit` system and DTrace. Robert Watson added an `audit provider` to the DTrace system in FreeBSD. A

DTrace provider gives access to a set of tracepoints from within DTrace. Some well-known examples of providers are those dealing with Function Boundary Tracepoints (fbt), System Calls (syscall), and network protocols (tcp, udp, ip). The audit provider gives DTrace the ability to record information about audit events that occur on the system while also applying DTrace features, such as filtering events via predicates and collecting statistical information via aggregations.

Using the audit system in the absence of DTrace, we did not have a convenient way to write runtime analysis scripts that allowed us to more finely target the processes that we wanted to investigate. The audit system will target a particular process, but we wanted to be able to collect data only when that process took a particular action.

Consider a scenario where we want to see who is talking to a web server; we might decide that we only want to know about connections that are coming from a specific set of Internet addresses, perhaps because we know those addresses have already been identified as part of a botnet. With the Audit Provider we can write a simple D script that asks only for the audit events relating to connect(2), and then filter those events based on the IP addresses they contain. Performing this data reduction as close to the source of the information as possible not only reduces the overall load on the system, but it also reduces the amount of data a human analyst or software tool has to look at during the later analysis phase.

OpenDTrace: The Future of DTrace

Besides Illumos, two other operating systems projects have ported and adopted DTrace. FreeBSD has had a port of DTrace since 2008 and Apple's Mac OS since 2007, where it is also integrated into the `Instruments` performance-analysis tool. With the demise of Sun Microsystems in 2010, the development of DTrace split, with some work being done within Oracle, but much of it moved onto Illumos, the fully open-source follow-on to OpenSolaris. Both FreeBSD and Mac OS continued to bring in changes from Illumos, but these were, for the most part, bug fixes rather than large new features. All three groups have done what they could to share code among themselves, but there

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations, and offers friendly, knowledgeable support staff. Starting at just $20/mo you are granted access to the latest FreeBSD, full Root Access, and Private Cloud options.

www.rootbsd.net

are significant differences among all three kernels. DTrace, although it was written in a good, portable style, requires many specialized hooks into the operating system, and this has resulted in some level of code drift. For example, the FreeBSD version of DTrace works on both ARMv8 and ARMv7 processors, but this is not yet true on Illumos. As part of our CADETS work, we realized that we wanted to add new features to DTrace and also to further abstract the code from any particular operating system, which led us to create OpenDTrace.

The aim of OpenDTrace is to provide a single, unified upstream for DTrace code that can then be easily imported into other operating systems, including FreeBSD, Mac OS, and Illumos, as well as others. The approach is similar to that taken by OpenBSM [12] and OpenZFS [13]. With this unified code base in place, we can then add features that apply to all the downstream OS consumers of DTrace far more quickly than we do today. .

Acknowledgment

The authors thank Ripduman Sohan for comments that greatly improved the manuscript. ●

JONATHAN ANDERSON is an Assistant Professor in Memorial University of Newfoundland's Department of Electrical and Computer Engineering, where he works at the intersection of operating systems, security, and software tools such as compilers. He is a FreeBSD committer and is always looking for new graduate students with similar interests.

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking, and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.

ARUN THOMAS is a researcher at BAE Systems R&D and the principal investigator of the the Causal, Adaptive, Distributed, and Efficient Tracing System (CADETS) project.

DR ROBERT N. M. WATSON is a Senior Lecturer (Associate Professor) at the University of Cambridge Computer Laboratory, where he leads research spanning operating systems, security, and computer architecture. He is a FreeBSD developer, member of the FreeBSD Foundation Board of Directors, and coauthor of *The Design and Implementation of the FreeBSD Operating System* (second edition).

This work has been sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

REFERENCES

[1] "Jails: Confining the omnipotent root." Poul-Henning Kamp <phk@FreeBSD.org> Robert N. M. Watson <rwatson@FreeBSD.org> (2000)

[2] Watson, R.; Feldman, B.; Migus, A.; and Vance, C. "Design and implementation of the Trusted BSD MAC framework," Proceedings—DARPA Information Survivability Conference and Exposition, DISCEX 2003, 1(Discex Iii), 38–49. http://doi.org/10.1109/DISCEX.2003.1194871 (2003)

[3] Watson, R. N. M. and Salamon, W. "The FreeBSD Audit System," UKUUG LISA Conference, 1–6. (2006)

[4] Watson, R. N. M.; Anderson, J.; Laurie, B.; and Kennaway, K. "Capsicum: practical capabilities for UNIX," 19th Usenix Security Symposium, (Figure 1), 3. http://doi.org/10.1145/2093548.2093572 (2010).

[5] Cantril, B.; Shapiro, M. and Leventhal, A. "Dynamic Instrumentation of Production Systems." (2004)

[6] Davis, B. *Everything you ever wanted to know about "hello, world"* (*but were afraid to ask)*, BSDCan. (2016)

[7] McKusick, M.; Neville-Neil, G.; and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System, Second Edition*. Boston, Massachusetts: Pearson Education. (2014)

[8] http://dtrace.org/blogs/bmc/2008/11/10/fishworks-now-it-can-be-told/

[9] https://www.splunk.com

[10] https://www.freebsd.org/cgi/man.cgi?query=audit&sektion=4

[11] https://www.niap-ccevs.org/pp/pp_os_ca_v1.d.pdf

[12] http://www.trustedbsd.org/openbsm.html

[13] http://open-zfs.org/wiki/Main_Page

SEE
TEXT
ONLY

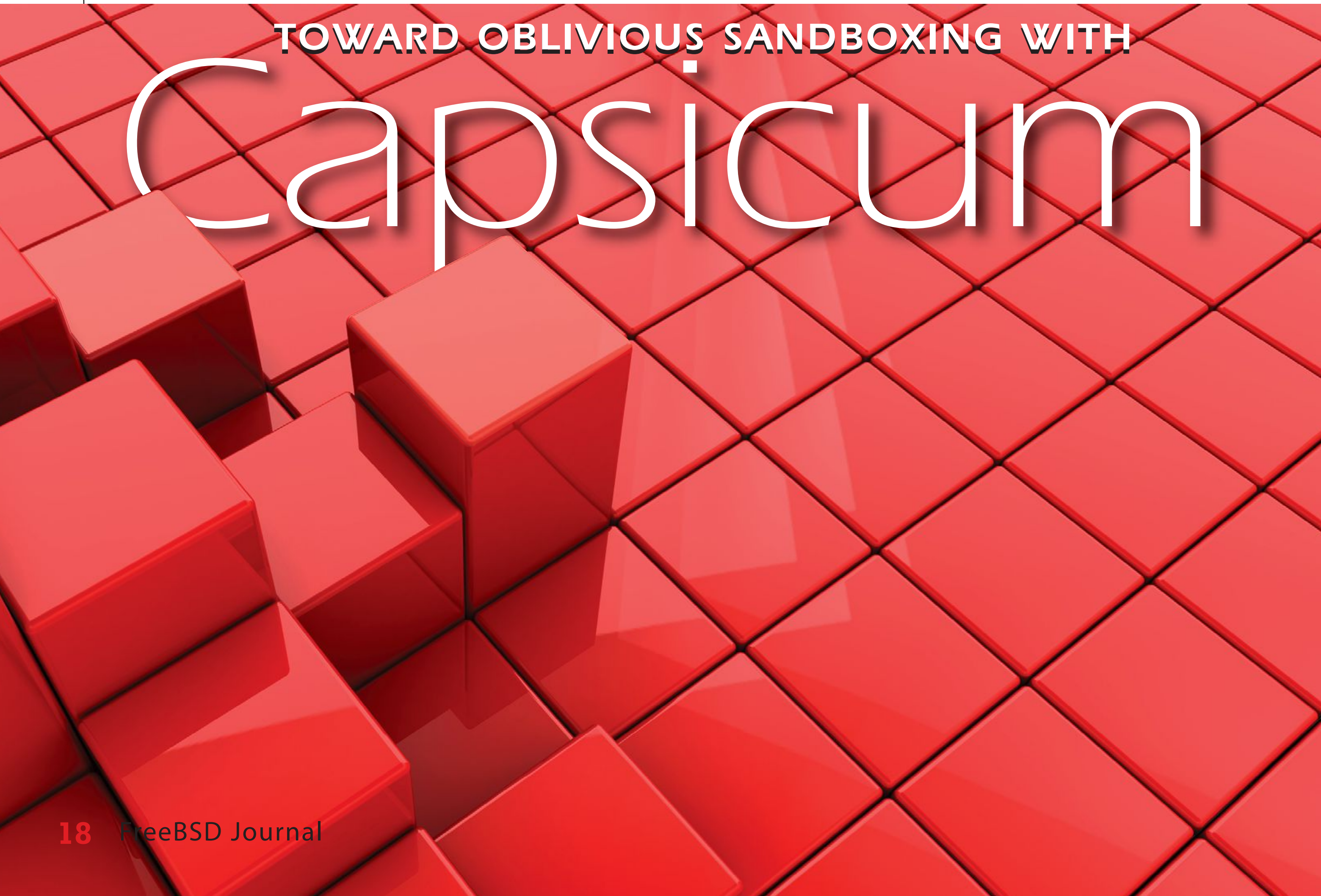
By Jonathan Anderson,
Stanley Godfrey, and Robert N. M. Watson

Capsicum [Wat+10] is a framework for principled, coherent compartmentalization of FreeBSD applications.

It is principled in that it draws from a rich history in computer security concepts such as *capabilities*, tokens that authorize their bearers to perform actions such as read from a file (using a file descriptor as a token very like a capability) or call a method (using an object reference as a capability). Capsicum is coherent in that it applies clear, simple security policies uniformly across applications. It is not possible—as can be the case in other schemes—to restrict an application's access to one set of operations while leaving equivalent operations available for use. When we describe Capsicum as providing principled, coherent *compartmentalization*, we mean that it allows applications to break themselves up into compartments that are isolated from each other and from other applications. Just as privacy-friendly companies put their users' data encryption keys out of their own reach, Capsicum allows applications and their compartments to give up certain abilities in order to protect other compartments, other applications, and—ultimately—their users.

However, a significant limitation of Capsicum today is that it only works when applications **voluntarily** give up the right to perform certain actions. It works with applications that understand Capsicum and that have been modified to take advantage of it; up to now, Capsicum has provided no mechanisms for confining applications without their cooperation. This is our long-term goal: to put applications into *sandboxes* without needing to modify the applications themselves, such that

TOWARD OBLIVIOUS SANDBOXING WITH Capsicum



any vulnerabilities in an application that are exploited by attackers can have their damage contained within an application's memory and outputs rather than granting full access to all of a user's data and activities. In this article, we describe recent and ongoing work to advance this agenda, pursuing the vision of protecting ourselves from vulnerable applications whether they like it or not.

A Taste of Capsicum

Today, Capsicum allows applications to protect themselves, other applications, and their users from themselves via two mechanisms: *capability mode* and *capabilities*.

Capability Mode

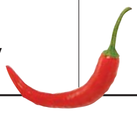
Capability mode is a way of *confining* a process to stop it from accessing any namespaces that are shared between processes such as the filesystem namespace, the process identifier (PID) namespace, socket-address namespaces, and interprocess communication (IPC) namespaces (System V and POSIX). Its only access to files and other system objects is mediated through *capabilities*, which are described below. Once a process enters capability mode, it loses all ability to access these namespaces and it cannot leave capability mode (nor can any processes forked from it from that point on). This first crucial Capsicum concept creates a strong isolation: if a process cannot open any resources *and holds no capabilities*, it cannot affect the operation of other processes or leave behind any side effects. For applications to do any useful work, however, some communications and/or side effects are necessary. The key, from a security perspective, is to ensure that these interactions occur in a **controlled** way.

Capabilities

The second crucial concept in Capsicum—the idea of *capabilities*—allows applications to be granted access to potentially-shared resources in a controlled way. Capabilities, as described by Dennis and Van Horn in 1966 [DV66], consist of an identifier or address for an object together with a description of the operations that may be performed on that object using the capability. In Dennis and Van Horn's model, computation occurred within a protection domain ("sphere of protection") and accessed resources using an index into a supervisor-maintained C-list. This model of userspace performing limited operations on system resources via indices within kernel-maintained arrays should sound familiar to current practitioners: capabilities were central to the PSOS design [FN79], which heavily influenced the design of Multics [SCS77], which directly inspired Unix and its *file descriptors* [RT78]. However, in the journey from capabilities to modern file descriptors, something was lost in translation: the rigorous, principled focus on capabilities as monotonic encodings of security policy (i.e., having a set of allowed operations that can be reduced, but never augmented). The Unix focus on user IDs within filesystems naturally led to an expansion of the role of file descriptors, such that the set of operations permitted via a file descriptor included operations that are not expressed in the descriptor itself, but are based on rights encoded in the filesystem. For example, most Unix-like systems will allow a user to `open(2)` a read-only file in a read-only mode and then `fchmod(2)` it to be a writable file (see Listing 1). This is an example of how file descriptors place more emphasis on the *identity* aspect of capabilities than on *operations*.

Capsicum capabilities restore to file descriptors a rigorous focus on allowed operations. In FreeBSD 10 and later, every file descriptor is associated with a set of explicit rights that define which operations may be performed on that file descriptor. Outside of capability mode, file descriptors are opened with all rights to preserve traditional file descriptor semantics. When descriptors are explicitly limited or derived from other capabilities (e.g., via `openat(2)` relative to a directory capability), only those operations explicitly permitted by the capability may be performed using that capability. There are capability rights that correspond to existing `open(2)` flags such as `CAP_READ` and `CAP_WRITE`, but there are also rights that make formerly implicit privileges a matter of explicit policy, such as `CAP_SEEK`, `CAP_MMAP`, `CAP_FTRUNCATE`, and `CAP_FCHMOD`.

Capabilities are monotonic: the holder of a capability may always give up rights associated with that capability with `cap_rights_limit(2)`, but new rights can never be added to an existing capability. If a process in capability mode requires access that it does not already have, it must acquire it from another process that rightfully has the authority to delegate it. Like all file descriptors,



```
int fd = open("my-data.dat", O_RDONLY);
if (fchmod(fd, 0777) < 0)
    err(-1, "unable_to_chmod");    // usually doesn't run!
```

Listing 1: File descriptors allow operations beyond those directly expressed in a descriptor—in this case, a read-only file descriptor is used to modify properties of the file.

capabilities may be delegated from one process to another via inheritance or IPC, but because of their strong monotonic guarantees, capabilities can be delegated with confidence: a capability with `CAP_READ` can be shared with an untrusted process in capability mode with the certain knowledge that it cannot be used to `fchmod(2)` the file or perform any actions other than `read(2)`. (The experimental implementation of capabilities in FreeBSD 9 involved additional indirection: a `struct` capability that contained a set of rights and a pointer to an underlying `struct` file.)

Why Capsicum?

Capsicum is a *principled* and *coherent* way to construct compartments within applications. It is principled in that it relies on a conceptually rigorous mechanism to enforce clear security policies that can be composed naturally due to the monotonicity of capabilities. As Linden observed in his 1976 survey of OS security and reliability [Lin76], "a single general protection mechanism that is used without exception is better than a rigid one that has many exceptions." Capabilities map naturally to many program requirements, as today's software is already structured around reference-like access to files as objects with explicit methods.

Capsicum's coherence is due to its "deep in the kernel" implementation and its simple-yet-complete definition of capabilities and capability mode. Attempts to provide "shallow" system call wrapping, exemplified by Provos's `systrace` [Pro03], are unable to provide the atomicity guarantees that are critical for security policy evaluation: policy enforcement is weakened when the objects and operations seen by the kernel are subject to races with the security policy that authorizes the operations on those objects. Attempts to allow userspace processes to define their own sets of "safe" system calls, as in Linux's `seccomp-bpf` [Cor12] or OpenBSD's `pledge(2)` mechanism [Chi15], can lead to incoherent security policies that disallow one type of access to system resources while permitting an equivalent type of access via another path. This can lead to an exposure of not-quite-sandboxed processes to malicious data under a false sense of security. In contrast, Capsicum's kernel-defined capability mode is both sufficient and necessary to express isolation from global OS namespaces, a coherent and easy-to-understand security policy.

The simplicity and dependability of capability mode helps application developers use it effectively, so long as their applications fit the simple model of first acquiring resources and then computing on them. (The section called Toward Oblivious Sandboxing on page 21 contains a discussion of more complex models.) Capsicum also requires no privilege for an application to compartmentalize itself, in contrast to approaches that rely on mandatory access control (MAC) such as SELinux [LS01] or AppArmor [BM06] or approaches that rely on Linux namespaces [Bie06]. Such approaches are accessible to applications with system-administrator support and/or `setuid` helper binaries, but Capsicum can be applied to any program compiled and run by any developer.

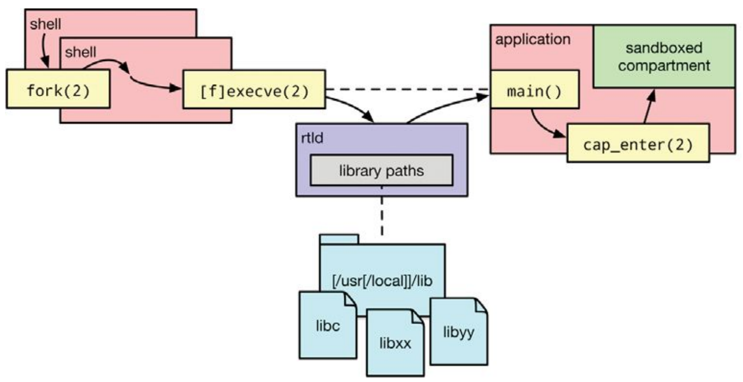


Fig. 1: Applications can compartmentalize themselves by acquiring static resources from global namespaces before calling `cap_enter(2)`.

Compartmentalizing with Capsicum

To take advantage of Capsicum, applications (including forked children of main application processes) call `cap_enter(2)` before they are exposed to any untrustworthy data, e.g., network requests. Once a process compartmentalizes itself, it can begin performing potentially dangerous operations such as parsing network traffic or user input in the confidence that any malicious exploitation will lead to, at worst, a corruption of the process's explicit outputs (files, network responses, etc.).

This self-sandboxing approach works well when a process is able to open all of the resources that it needs before entering capability mode. The most obvious resources to be opened before compartmentalization are files and sockets, but in a modern binary, even something as simple as `cat(1)` or `echo(1)`, dynamic linking means that a set of shared libraries must also be loaded before compartmentalization. As shown in Figure 1, the runtime linker runs within a process, sharing its address space and, on startup, its main thread. Most simple applications only rely on the runtime linker to find all their libraries on startup, after which it's possible to enter capability mode and let the runtime linker fix up dynamic symbols as required from already-open library files.

Other static resources that a sandboxed compartment might need to access include explicit files, which can be pre-opened by an application before calling `cap_enter(2)` or accessed via a pre-opened *directory descriptor* and then accessed by the compartment with `openat(2)` and related system calls (`fstatat(2)`, `renameat(2)`, etc.). Implicit resources include locale files that are required by many libc functions, but these can also be pre-opened and their results cached. More dynamic resources require a connection with the outside world so that a sandboxed process can ask an unsandboxed process to access new resources. This mode of operation is commonly employed in compartmentalized applications on many platforms, including Web browsers and—perhaps surprisingly—all applications downloaded on MacOS through the Mac App Store, where all applications requesting access to user files must go through a trusted UI called a *powerbox* [App16; Yee04]. To help with these more dynamic applications' requirements, FreeBSD includes the `libcasper` (Capability Services Provider [Zab16]) mechanism to proxy access to named services, some of which (e.g., `system.dns`) are provided by the system itself.

With pre-opened capabilities, locale cacheing, directory descriptors, libcasper, and external proxies at their disposal, many applications are able to compartmentalize themselves with Capsicum. However, this only applies to applications whose authors are willing to spend the effort required to adopt Capsicum features and adapt their applications for compartmentalization. More impact could be attained if we were able to transparently sandbox applications without imposing any additional requirements on their authors, i.e., if we could employ *oblivious sandboxing*.

Toward Oblivious Sandboxing

With this goal of oblivious sandboxing in mind, work on Capsicum has been progressing across FreeBSD's runtime linker, a new library, and a new capability-aware (but not feature-complete) shell. Recently, some of these components have begun to bear fruit, leading to an exciting new development: the first transparent sandboxing of unmodified applications within Capsicum's capability mode. The applications that can be executed today in this matter are very simple, but they execute without access to global namespaces and without any modification: rather than sandboxing themselves, they begin life in a sandbox.

`exec(2)` Without a Name

The traditional approach by which one application executes another is to first `fork(2)` a child process and then, from within that new process, to call `exec(2)` and start running the new program. The `exec(2)` system call cleans up the memory mappings of the current process, closes any file descriptors that have an `O_CLOEXEC` flag set (preserving all other open files, together with current environment variables) and transfers control to the new application. In order to do this, `exec(2)` must first find the binary to be executed. Looking up a binary by name—as in the traditional `exec(2)` call—would require access to the global filesystem namespace; this is not permitted in capability mode. Instead, FreeBSD provides the `fexecve(2)` system call to execute a binary as specified by a file descriptor (which can be a capability) rather than by pathname. On Linux, `fexecve(3)` is a `glibc` function that uses a file descriptor to look up a symbolic link in `/proc/self/fd`, then calls `exec(2)` with that path name. The Linux implementation of Capsicum required the addition of an `execveat(2)` system call with true file descriptor semantics [Dry14].

When `fexecve(2)` runs, it inspects the file passed to it to determine its type (ELF executable, script, `a.out` executable, etc.) and passes it to an *image activator* within the kernel (see Figure 2). Image activators parse various types of executable files and start running them; ELF image activators (32- and 64-bit) encode knowledge of runtime linkers and how to find them in the filesystem.

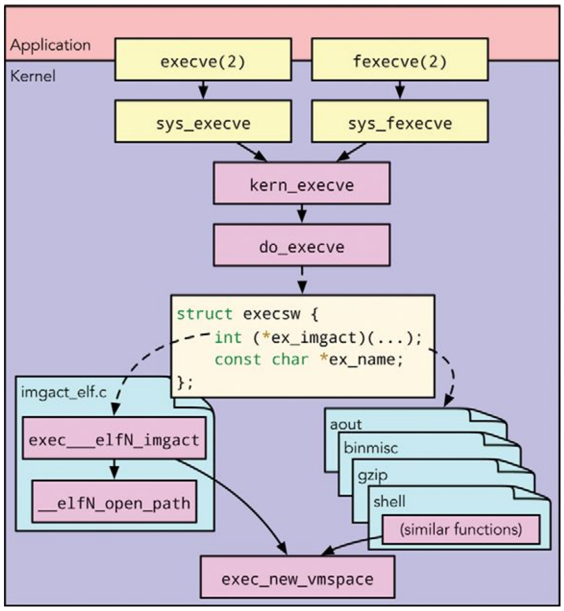


Fig. 2: The FreeBSD kernel contains a number of *image activators* to parse various types of executable files and start running them.

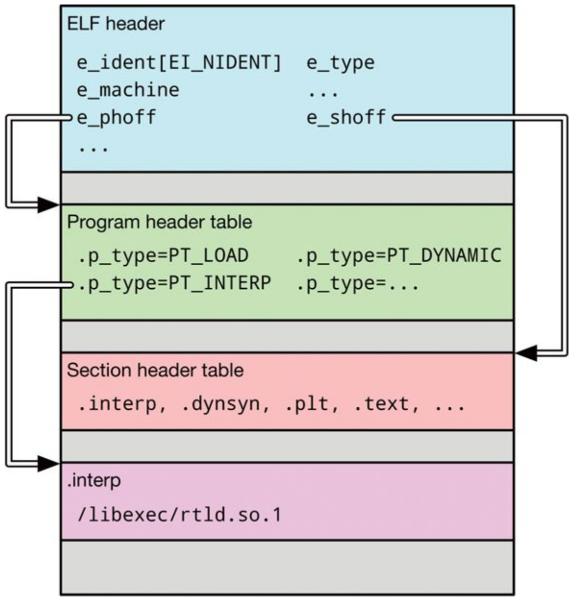


Fig. 3: The ELF file format includes an explicit path to the runtime linker that is expected to be used as an interpreter for the binary file.

Although various ABIs for various platforms have default runtime linker names, binaries can also explicitly encode a path to a preferred runtime linker, as shown in Figure 3. Whether discovered via a program's internal format header or the default of an image activator, runtime linkers are described using path names. In a conventional `exec(2)` or `fexecve(2)` invocation, the runtime linker would be looked up using this path and executed first, before the main function of the new application (illustrated in Figure 1). Inside a Capsicum compartment, however, access to global filesystem namespaces is not permitted, so another approach is required.

FreeBSD's runtime ELF linker has recently been modified to support direct execution, i.e., on FreeBSD 12-CURRENT one can run `/libexec/ld-elf.so.1` as an executable—the usage string as of writing is shown in Listing 2. This ability has long been present in Linux's `ld-linux.so.2`, but it was not required on FreeBSD until motivated by the requirements of oblivious sandboxing. Now, direct execution has been implemented together with the ability for the runtime linker to accept as a command-line argument a file descriptor to link and run—these changes will be present in FreeBSD 12 and 11.1. Together, they allow a process in capability mode that has capabilities for a runtime linker and a binary to `fexecve(2)` the linker, preserving open files including the binary's capability, and to specify via command-line arguments which file the linker is to execute. The net result, shown in Figure 4, is that the specified binary is executed using the specified runtime linker. However, without access to shared libraries stored in the filesystem, the runtime linker is not able to satisfy the dynamic code-loading requirements of the application. That requires an additional mechanism: library path descriptors.

Shared Libraries in Capability Mode

As described previously, essentially all modern executable files are dynamically linked and therefore depend on access to shared libraries for their correct execution. In fact, the FreeBSD-derived MacOS does not support statically linked binaries: ABI guarantees are maintained only at the interfaces of core system libraries rather than the kernel [App11]. When applications compartmentalize themselves with `cap_enter(2)`, they can do so after the dynamic runtime linker has discovered library dependencies and `mmap(2)`'ed them in place for later linking. If an application starts running before these libraries have been opened, however, the linker is unable to satisfy the requirements of dynamic symbol resolution.

Traditionally, the dynamic runtime linker has supported a number of environment variables that control its behavior. One example, `LD_LIBRARY_PATH`, informs the linker of a set of directories in which additional libraries may be found. For example, a program may set `LD_LIBRARY_PATH` to an internal directory that contains application code or dynamically-loadable plugins. We have extended FreeBSD's ELF runtime linker to support an additional environment vari-

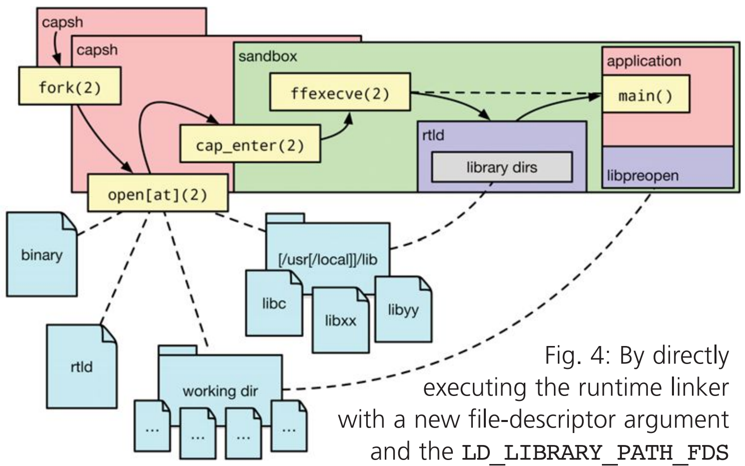
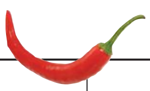


Fig. 4: By directly executing the runtime linker with a new file-descriptor argument and the `LD_LIBRARY_PATH_FDS` environment variable, `capsh` can execute an untrusted program from within a Capsicum sandbox. This application starts running without the ambient authority to access global namespaces.

```
Usage: /libexec/ld-elf.so.1 [-h] [-f <FD>] [--] <binary> [<args>]
Options:
  -h         Display this help message
  -f <FD>    Execute <FD> instead of searching for <binary>
  --         End of RTLD options
  <binary>   Name of process to execute
  <args>     Arguments to the executed process
```



Listing 2: Current usage options for FreeBSD's ELF runtime linker.

able, `LD_LIBRARY_PATH_FDS`. This variable allows the specification of directories containing shared libraries that will be searched in exactly the same way as `LD_LIBRARY_PATH_FDS`, but with one crucial difference: instead of a colon-separated list of pathnames, this variable contains a colon-separated list of directory descriptors. Since environment variables and open files are both preserved across the `fexecve(2)` boundary—unlike, e.g., memory mappings—this allows a parent process to open a set of library descriptors, set `LD_LIBRARY_PATH_FDS` and then enter capability mode and `fexecve(2)` the runtime linker itself with access to its shared library directories. Combined with the direct-execution support described in the previous section, this allows a dynamically linked application to be executed from within a sandbox.

`libpreopen`: Transparent Filesystem Proxying

It is useful to be able to execute code from within a sandbox, including code that is dynamically linked, but that is insufficient for the goal of oblivious sandboxing. Most Unix applications are written using common system calls such as `access(2)`, `stat(2)`, and `open(2)` to test and gain access to files within the filesystem. These system calls inherently require access to the global filesystem namespace, so they are not permitted in capability mode. It is possible to write applications to use `fstatat(2)`, `openat(2)`, etc., relative to an explicit base directory, but many extant applications have not been written this way. To achieve oblivious sandboxing, applications must be confined and resources must be provided *without* application modification.

We can interpose a runtime translation of system calls from capability-unsafe to capability-safe variants using the runtime linker: the `LD_PRELOAD` environment variable allows us to name libraries that should be loaded before any others. When libraries are named in this environment variable without absolute paths, the runtime linker searches through its default search paths for libraries of the given names, but not before consulting `LD_LIBRARY_PATH_FDS`, making `LD_PRELOAD` a capability-mode-compatible directive. If we provide an implementation of a `libc` function such as `open(2)`, our implementation will take precedence over that of libc, where system calls are defined as "weak" symbols. Our implementation of `open(2)` can translate the provided path argument into an `openat(2)` call, but by itself, this adaptation accomplishes nothing: the application will still attempt to look up a path name that is not relative to a directory, only this time it will do so using the `openat(2)` system call instead of `open(2)`.

The final component that is required to adapt filesystem namespace operations is a set of pre-opened directory descriptors that other operations can be performed relative to. This is the core abstraction provided by `libpreopen`, a library that is—for the moment—maintained independently of FreeBSD. (`libpreopen` can be downloaded and built from https://github.com/musec/libpreopen). `libpreopen` provides a `struct po_map` type that is used to map directory names to directory descriptors, flags, and capability rights, as well as `libc` wrappers that can look up and query a default `po_map`. For example, when `libpreopen`'s implementation of `open(2)` is passed an absolute path, it looks up the default `po_map`, which can be specified as data packed into an anonymous shared memory segment. FreeBSD's implementation of POSIX shared memory allows a constant "path" of `SHM_ANON` to be passed to `shm_open(2)`, creating a shared memory segment that can be manipulated by file descriptor but does not appear in the regular POSIX shared memory namespace, making it safe for use in capability mode. `libpreopen` can open such a shared memory segment, specified by file descriptor in an environment variable, and unpack its data into an in-memory `struct po_map` object. From that `po_map`, the wrapper queries, "do you have a directory descriptor whose name is a prefix of this absolute path?" If such a descriptor exists in the map, the absolute path is decomposed into a directory descriptor and a relative path from that descriptor. These two elements can then be passed to `openat(2)`.

`libpreopen` provides a mechanism for a process in capability mode to access filesystem resources, as long as some directory descriptors have been pre-opened and stored in a way that is accessible to the library. Opening such descriptors, building a `struct po_map` representation, packing it into anonymous shared memory, and storing the file descriptor of the shared memory segment in an environment variable are all the responsibility of the process spawning the sandboxed child. One example of such a process is **capsh**, the capability shell.

capsh: A Capability-enhanced Shell

The final major component of Capsicum-based oblivious sandboxing is a program to transparently sandbox unmodified—and unsuspecting—applications. A proof-of-concept implementation of such a program is **capsh**, a shell that uses capabilities and capability mode to sandbox applications. Hosted independently of FreeBSD for the time being, capsh allows users to execute simple unmodified applications from within a Capsicum sandbox. (The **capsh** source code is available at https://github.com/musec/capsh.) In its current implementation, the program is hardly a shell at all: it has no interactive mode, only executing a single program per invocation. It also only supports simple applications with statically-enumerable resource requirements. Nonetheless, programs that fit into this model can be executed with sandboxing from inception without program modification.

capsh works by tying together the pieces of the oblivious sandboxing puzzle described above. It finds and opens a user-specified executable file, together with a runtime linker to interpret it. It opens library directories and stores them in the **LD_LIBRARY_PATH_FDS** environment variable. It manipulates pre-opened directory descriptors, storing them in **struct po_map** types provided by **libpreopen** and making them available to child processes via shared memory and environment variables. It then enters capability mode via **cap_enter(2)** and uses **fexecve(2)** to execute the runtime linker. The net result is that an unmodified application starts running from within a Capsicum sandbox, as shown in Figure 4.

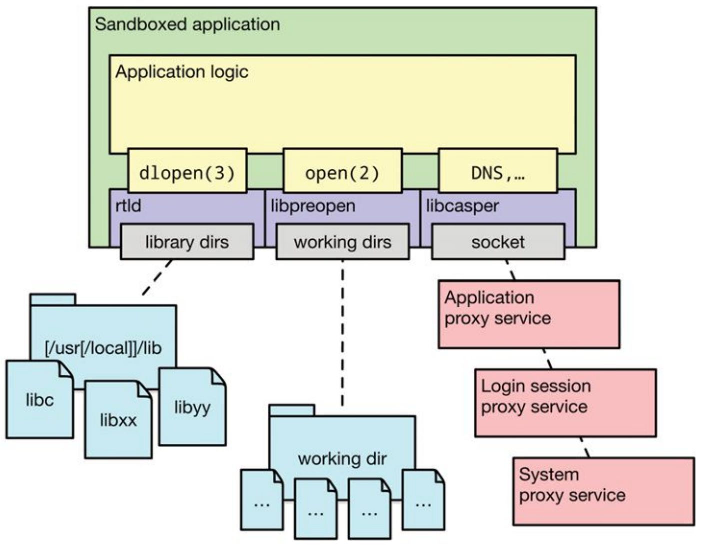


Fig. 5: A fully-sandboxed application can only access files and services that have been delegated to it: the runtime linker can find libraries in pre-opened directories using `LD_LIBRARY_PATH_FDS`, **libpreopen** can operate on files in pre-opened working directories—translating global-namespace–dependent system calls such as `open(2)` to relative variants such as `openat(2)`—and **libcasper** can proxy access to namespaces of external servers.

Oblivious Sandboxing

Applications running under capsh can access only those resources that are explicitly delegated to them; there can be many sources of policy as to which resources ought to be delegated. Users running **capsh** implicitly specify policy when they type command-line arguments to sandboxed applications: the presence of a filename as an argument may be an indication that the file should be pre-opened before the application is executed or that permission to open the file via a proxied mechanism such as **libcasper** should be granted. Users may also drive policy decisions implicitly through interactions with a graphical user interface, as in the powerbox model described in the earlier section on Compartmentalizing with Capsicum, and future work on **capsh** may connect to existing models of graphical login sessions to provide this mode of policy elicitation. Policy may also be derived from files packaged together with applications: a compiler's package metadata may specify where its standard library is located, and **capsh** could pre-open that directory with a read-only capability. More sophisticated policy files could describe limited interactions that are permitted with named libcasper services, leading to a more general model of a Capsicum application as shown in Figure 5. Additional exploration of mechanism is also possible: of particular interest to the authors of this article is the possibility of applying LLVM-based transformation to libraries and applications that embed LLVM bitcode, allowing a transparent rewriting of function calls to capability-mode–friendly APIs without needing `LD_PRELOAD` for interposition.

Conclusion

Capsicum, a principled and coherent design for software compartmentalization, has taken strides in recent days toward a new security model. Changes in the FreeBSD ELF runtime linker, together with developments in **libpreopen** and **capsh**, allow simple applications to be sandboxed *transparently*, without any participation on the part of the application. These foundational elements have now set the stage for a deeper exploration of how programming models interact with the need for compartmentalization and to what extent software can be sandboxed *obliviously*, operating as normal with no requirement to know whether it is operating inside of a sandbox or not. A broader availability of oblivious sandboxing will allow us to move to a FreeBSD in which applications "just work" *and* are secure by default. •

This work has been sponsored by the Research & Development Corporation of Newfoundland & Labrador (contract 5404.1822.101), the NSERC Discovery program (RGPIN-2015-06048), and the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Jonathan Anderson is an Assistant Professor in Memorial University of Newfoundland's Department of Electrical and Computer Engineering, where he works at the intersection of operating systems, security, and software tools such as compilers. He is a FreeBSD committer and is always looking for new graduate students with similar interests.

Stanley Godfrey is a graduate student at Memorial University of Newfoundland. His research interest is in Capsicum, FreeBSD, and operating system security. He did his undergraduate studies at Helsinki Metropolia University of Applied Sciences, majoring in software development and graduating with a Bachelor of Engineering in Information Technology.

Dr. Robert N. M. Watson is a Senior Lecturer (Associate Professor) at the University of Cambridge Computer Laboratory, where he leads research spanning operating systems, security, and computer architecture. He is a FreeBSD developer, member of the FreeBSD Foundation Board of Directors, and coauthor of *The Design and Implementation of the FreeBSD Operating System (second edition)*.

[App11] Apple Inc. "Technical Q&A QA1118: Statically linked binaries on Mac OS X," *Apple Developer Guides*. https://developer.apple.com/library/content/qa/qa1118. (2011)

[App16] Apple Inc. "App Sandbox in Depth," *Apple Developer Guides*. https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxInDepth/AppSandboxInDepth.html. (2016)

[Bie06] Biederman, Eric W. "Multiple Instances of the Global Linux Namespaces," *Linux Symposium Volume One*, pp. 101–111. https://www.landley.net/kdocs/ols/2006/ols2006v1-pages-101-112.pdf. (2006)

[BM06] Bauer, Mick. "Paranoid Penguin: An Introduction to Novell AppArmor," *Linux Journal* 2006.148, p. 13. ISSN: 1075-3583. URL: https://dl.acm.org/citation.cfm?id=1149839. (2006)

[Chi15] Chirgwin, Richard. "Untamed pledge () aims to improve OpenBSD security: Monkey with the wrong permissions, your program dies," *The Register*. https://www.theregister.co.uk/2015/11/10/untamed_pledge_hopes_to_improve_openbsd_security. (2015)

[Cor12] Corbet, Jonathan. "Yet another new approach to seccomp." https://lwn.net/Articles/475043/. (2012)

[Dry14] Drysdale, David. "syscalls,x86: Add execveat() system call," *Linux Kernel Mailing List*. https://lkml.org/lkml/2014/5/27/147. (2014)

[DV66] Dennis, Jack B. and Van Horn, Earl C. "Programming semantics for multiprogrammed computations," *Communications of the ACM* 9.3, pp. 143–155. DOI: 10.1145/365230.365252. (1966)

[FN79] Feiertag, R. J. and Neumann, Peter G. "The foundations of a provably secure operating system (PSOS)," NCC '79: *Proceedings of the 1979 AFIPS National Computer Conference*. DOI: 10.1109/AFIPS.1979.116. (1979)

[Lin76] Linden, Theodore. "Operating System Structures to Support Security and Reliable Software." ACM Computing Surveys (CSUR) 8.4, pp. 409–445. DOI: 10.1145/356678.356682. (1976)

[LS01] Loscocco, Peter A. and Smalley, Stephen D. "Meeting Critical Security Objectives with Security-Enhanced Linux," *Proceedings of the 2001 Ottawa Linux Symposium*. https://lwn.net/2001/features/OLS/pdf/pdf/selinux.pdf. (2001)

[Pro03] Provos, Niels. "Improving Host Security with System Call Policies," *Proceedings of the 12th USENIX Security Symposium*. http://niels.xtdnet.nl/papers/systrace.pdf. (2003)

[RT78] Ritchie, O. M. and Thompson, K. "The UNIX time-sharing system," *Bell System Technical Journal* 57.6, pp. 1905–1929. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02136.x. (July 1978)

[SCS77] Schroeder, Michael D.; Clark, David D.; and Saltzer, Jerome H. "The Multics kernel design project," SOSP '77: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*. ACM. DOI: 10.1145/800214.806546. (1977)

[Wat+10] Watson, Robert N. M.; Anderson, Jonathan; Laurie, Ben; and Kennaway, Kris. "Capsicum: practical capabilities for UNIX," *Proceedings of the 19th USENIX Security Symposium*. https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf. (2010)

[Yee04] Yee, Ka-Ping. "Aligning security and usability," *IEEE Security and Privacy Magazine* 2.5, pp. 48–55. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.64. (2004)

[Zab16] Zaborski, Mariusz. "libcasper(3)," *FreeBSD Library Functions Manual*. https://www.freebsd.org/cgi/man.cgi?query=libcasper. (2016)

SEE TEXT ONLY

GAMING BSD

IF YOU HAVE BEGUN READING THIS, IT PROBABLY MEANS YOU ARE INTERESTED IN SECURITY OR IN EMBEDDED SOFTWARE OR YOU WORK IN THE GAMING BUSINESS. IN ANY CASE, PLEASE PUT ASIDE YOUR ANTIPSYCHOTIC MEDICATION RIGHT NOW, BECAUSE WE NEED YOU IN YOUR MOST PARANOID STATE, READY TO WEAR THE TWO HATS OF SECURITY, THE BLACK ONE AND THE WHITE ONE.

SECURITY has been discussed a lot over the years, particularly when focusing on protecting customers' data. That data can be simple data, such as an email address or a user-name, or more elaborate data like a password used to decrypt the shopping done by a customer in an online shop. These services are usually based on data protection and do not ensure that the hardware or software being used is what is actually delivered. In the case of servers, it is useless to talk about hardware protection because only the system administrator should have physical access to those machines.

In the gaming industry, we have software and hardware, both of which must be checked to ensure that the delivered product has not been manipulated. For example, manipulation could be done by changing some of the product's hardware components or by inserting a way to bypass the protection mechanism implemented on it.

In addition to a hardware intrusion detection system, it is highly recommended that the software developed for the product also have some kind of protection. FreeBSD offers several tools for that, starting with encrypting the disk on

By Roberto Fernández

which the programs are installed and ending by detecting whether they were somehow changed.

We should not forget that not every single piece of code will be used in a gaming system. For instance, `freebsd-update(8)` could be used or not. If you are not using something, it could be a security risk for the system if it is left on a release version. If you do use it, then you have to control which security issues affect your software and how patches could be applied to the system.

But there is more to it than just protecting hardware or software. As a gamer, I am only concerned that the data I saved do not get lost. When I save the progress I made on a game, I would hate to see that get lost and require me to redo what I had already done. Imagine that you have finally gotten over a scene that took maybe five hours to get through, and then the saved point was lost. Would you be "happy" or would you want to tear apart your console?

BUILDING IT AS EMBEDDED

The best way to make sure your software has as few security holes as possible is to use YAGNI (You aren't going to need it). Although this is a technique for writing high-quality code, it could be used here to inform you if a program or a library is not going to be used and thus keep you from including it.

To get a better understanding of how this is accomplished, the building section is divided into three parts: building the world, building the kernel, and building ports.

Building the World

There are two ways to accomplish this:

- Build everything and install something.
- Build only what you need.

The first axiom implies a full build with well-known commands:

```
root@ASUS-R752L: /usr/src # make buildworld
root@ASUS-R752L: /usr/src # make installworld
     DESTDIR=/work/target
```

And after its completion, all the items go into the final system with a shell script. This is a way to start if you do not want to mess around with the FreeBSD build process. An optimization technique for this process is to write a `src.conf(5)` file and set the SRCCONF variable in your environment. By doing that, your build process could boost up a lot, depending on how many of the settings are left out of the build process.

The second axiom requires the development of a build variable like the ones from `src.conf(5)`. Explaining this fully would require an entire article in and of itself, and it is not the main focus here. With proper implementation of the variable, the build process could be sped up by 50% and thus avoid long waiting times.

Building the Kernel

Inside the kernel configuration file, there are a lot of devices and options that might never get used on your machine. Examples are NFS options or RAID devices. Again, if you do not need them in your kernel, a new kernel's configuration file should be written without them in order to get the kernel to run faster and, in case someone wants to add hardware to the final product, to reduce the security risk if it is not supported.

The kernel has already been configured to run faster and contains support for the devices you want, but that is not all there is to it. The boot partition must not be encrypted (the last time I checked, Allan Jude was not done with the geli boot loader, at least in an automatic way to boot without typing the password) and the `loader.conf(5)` is saved there, which allows an attacker to modify it and get support for the hardware they want.

Building Ports

This is the trickiest question of all. We can build the operating system in any version of FreeBSD without messing around with your own build system. But this does not work with ports. If you want to get a set of programs or libraries from the ports tree and install them into your product, you need to do something else.

There are several techniques for this. The one used to create packages that will be used by the final user is `ports-mgnt/poudriere`. This tool creates a jail and builds the ports there using the same version of the base system. After building the ports and staging them, you should write a script that is able to get only the libraries and binaries you want. A simple way is to write a file like the `pkg-plist` which is on most of the ports and describes which components must be installed on the real system. Then the script mentioned before should read this file, fetch the libraries and binaries, and copy them into your target bintree.

Right now, you have all you need to run your gaming system, but there are a lot of security reports for the programs that might be installed on the product. They should be tracked down to ensure that they do not affect your software and that your product will not be compromised.

WRITING ATOMIC DATA

As explained previously, the data we write on disk must be consistent, which means that the writing must not be in an idle state where the data to be read will be shown as corrupted and no longer usable. There are several causes of data loss. The first, and the most common in the gaming industry, is hard shutdown. This means that the system is not shut down by software, but by hardware which causes the data to not be entirely written on the disk.

FreeBSD has developed a way to ensure that the data is consistent by writing it entirely or not at all. The way to do this is `gjournal(8)`. Usually, it is easier and more comfortable to write the data needed by the user in a non-encrypted device or partition. That way the user can back up saved games and a gamer profile, or copy this information to bring it into another system. Also, the support team can back up user data before starting a reparation.

For this example, we will use a 1-TB hard disk that contains only user data. The GEOM partition scheme should look like the following:

```
root@ASUS-R752L: ~ # gpart show -1 ada1
```

```
        34  1953525101  ada1         GPT  (932G)
        34           6         - free -   (3.0K)
        40  1953525096     1  UserData   (931G)
1953525136           5         - free -   (2.5K)
```

In order to put some journaling into the `gpt/User-Data` partition, which is set in a system with 8 GB of physical memory, the following commands must be run:

```
root@ASUS-R752L: ~ # gjournal label -s 16G \
     gpt/UserData
root@ASUS-R752L: ~ # ls /dev/ufs/UserData*
UserData UserData.journal
root@ASUS-R752L: ~ # newfs -J -L UserData
     /dev/gpt/UserData.journal
```

After that, the `/dev/ufs/UserData` entry should be written in the `fstab(5)`. It is important to use that one and not the GEOM identifier or the driver name, because it can cause the wrong partition or device to be mounted.

PROTECTING VITAL DATA

One of the biggest worries in the embedded industry is that proprietary software gets modified or its behavior discovered. This is done to bypass authentication methods implemented to deny the execution of unknown software that allows an attacker the use of modified data or, in the case of the gaming industry, to play an illegal copy of your software.

A simple way to protect your data is to encrypt it with the cryptographic GEOM class, also known as `geli(8)`, where you select the programs to be protected. You can encrypt the whole system or only a partition where your programs are going to be stored, so that in boot time, you can check if any manipulation took place and decrypt this partition if none have been taken. In either case, you have to find a secure place to store your key. It can be generated in runtime or stored in a chip connected to your system, which, under certain circumstances, will give you back the key.

The Trusted Computing Group has developed a standard for this purpose—The Trusted Platform Module or TPM, which is a chip that has, among other things, a non-volatile memory where you can write information and seal it with values stored in the registers. Usually, the BIOS will calculate its own checksum and the MBR's and store them in the TPM's registers. After the system has completely booted, the other registers can be written with some calculated data to have access to the information stored in the chip's memory which holds the key for decrypting the root partition.

If your platform is using a UEFI boot, then the Secure Boot is an additional step that could be used, allowing the machine to have a signed boot process and to look for a manipulation on the boot chain.

AUTHENTICATING PROPRIETARY SOFTWARE

The authentication of proprietary software comes after the machine has completely booted. The simplest way to do this is to write in a file which files must be checked and which signature they must have, the same thing FreeBSD does with release files (installation media).

Another way to do it is to write a program that reads from a binary file using a defined protocol and checks whether the programs were somehow altered. This version requires a thorough understanding of authentication methods and how to calculate checksums of binary files.

But what is interesting right now is not how to detect a manipulation in your software, but what to do when you detect one. As an example, Microsoft® bans the users who tried to use a backed-up copy of a game in a Xbox 360® from the online service. It prohibits the users from updating the system or downloading games using that account. Thus it leads the user to create a new one and replay all the games again in order to restore the user's online status as gamer.

That is ideal, but if what has been altered is how the system works, then it is a good idea to "forget" the key to decrypt the right partition, so that the user will have a real expensive paperweight that will never boot again. It is worthwhile to stop and consider the severity of the manipulation and act accordingly. For example, it is not a good idea to destroy the decryption key when the user tries to play an illegal copy of your games when the worst thing that can be achieved is playing the game. On the other hand, if the manipulation could cause the system to be used as a fraud machine, then that option is the right call.

DETECTING HARDWARE MANIPULATION

When installing the product, information about the system can be obtained and stored on a file that will be read after the system has booted to check whether your hardware has changed. The list of hardware attached to your machine and where it is attached can be obtained by running `pciconf(8)` and `usbconfig(8)`, so you can have control of which components are connected via a PCI card and which ones are attached via USB ports or HUBs.

Let's consider an example. The first of the following scripts will get the information gathered by `pciconf(8)` and `usbconfig(8)` and store it in plain text under `/var/db/hw` with the names `pci.db` and `usb.db`. The second will get the information and compare it with the information stored in those files. To get a better understanding of the process, let's remove a security layer (encryption, authentication, or other such mechanisms) from the file and it will be stored in plain text.

```
#!/bin/sh

[ ! -d /var/db/hw ] && mkdir -p /var/db/hw
pciconf -l | sort | cut -d '@' -f1 |\
   while read DEVICE ; do
      echo ${DEVICE} | grep -q 'none' && continue

      pciconf -lv ${DEVICE}
   done > /var/db/hw/pci.db

usbconfig | cut -d ':' -f1 | sort | \
   while read DEVICE ; do
      usbconfig -d ${DEVICE} dump_device_desc
   done > /var/db/hw/usb.db
--------------------------------------------
#!/bin/sh

pciconf -l | sort | cut -d '@' -f1 |\
   while read DEVICE ; do
      echo ${DEVICE} | grep -q 'none' && continue

      pciconf -lv ${DEVICE}
   done > /tmp/pci.db

usbconfig | cut -d ':' -f1 | sort | \
  while read DEVICE ; do
      usbconfig -d ${DEVICE} dump_device_desc
   done > /tmp/usb.db

test "$(comm -3 /tmp/pci.db /var/db/hw/pci.db)"
[ ${?} -eq 0 ] && exit 1
test "$(comm -3 /tmp/usb.db /var/db/hw/usb.db)"
[ ${?} -eq 0 ] && exit 1

exit 0
```

If a TPM chip is installed on your product or the UEFI Secure Boot has been selected, your BIOS has already been checked and therefore there is no need to check it again. The port `sysutils/dmidecode` offers the option of checking the vendor and version of the BIOS or your motherboard if there is a SMBIOS or DMI entry point. If `dmidecode(8)` does not work for you, then you should find a way to read this information as it will be worth it in the long run.

The machine's processor is another thing that is worth checking, and `sysctl(8)` is good enough for this. By checking the node `hw.model`, the model of the CPU is returned, allowing the product to check its own processor and boot when it was not changed at all.

INTERFACES

Up till now, we have talked about product. This section explains the risks in connection with any interface to the external world—for instance, the network card. There are several known variations of attack attempts, especially with regard to communications with the external world. Let's consider them through the interfaces.

Network

Today, in the IoT (Internet of things) era, it is normal to communicate a system to the Internet, and it does not matter if it is a toaster or a server. This is a security risk for your product, and some thought is required before connecting a device to the external world. The question to ask is: why must it be connected to the Internet and which services should be allowed?

The product can be connected to the Internet because there might be an online multi-player version of your game or an update service required by the system to keep it up-to-date with the most secure version of the basic software. If only the first condition is relevant, then the product should forbid the user from using other services. But if the product allows the user to have an HTTP browser, then the browser must be configured to deny pages that may compromise the product.

Another issue is the "man in the middle" problem, which asks how you secure your channel to avoid the communication from being sniffed. Will it use IPsec, SSL, TLS, or are you implementing a new cryptographic layer for your communication?

I cannot really say which is better, because each has advantages and disadvantages. It is the mission of the system developer to evaluate the needs of the final product and to set the most secure configuration.

USB

USBs are used for almost everything today. There are adapters for hard disks, Ethernet or wireless connections, webcams, touch screens, and serial communications. One might think that the USB is secure enough, but it depends on which upper layer protocol you are using. If a program using the `libusb(3)` library or a kernel driver is developed, it must be good enough to avoid memory leaks or code injection.

Others

No two systems are identical, so I am not in a position to say which interfaces should be protected and which not. But be paranoid, go through the drivers that are needed, search for security flaws, try to hack them and improve the drivers. FreeBSD can use the help of system developers working in the gaming industry to make it safer!

ROBERTO FERNÁNDEZ lives in Berlin, Germany, where he has worked in the gaming industry for two years and programmed with FreeBSD for three years. When not spending time with his son and wife, he plays video games.

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with BSD CERTIFICATION

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification **YOU'LL HAVE WHAT IT TAKES!**

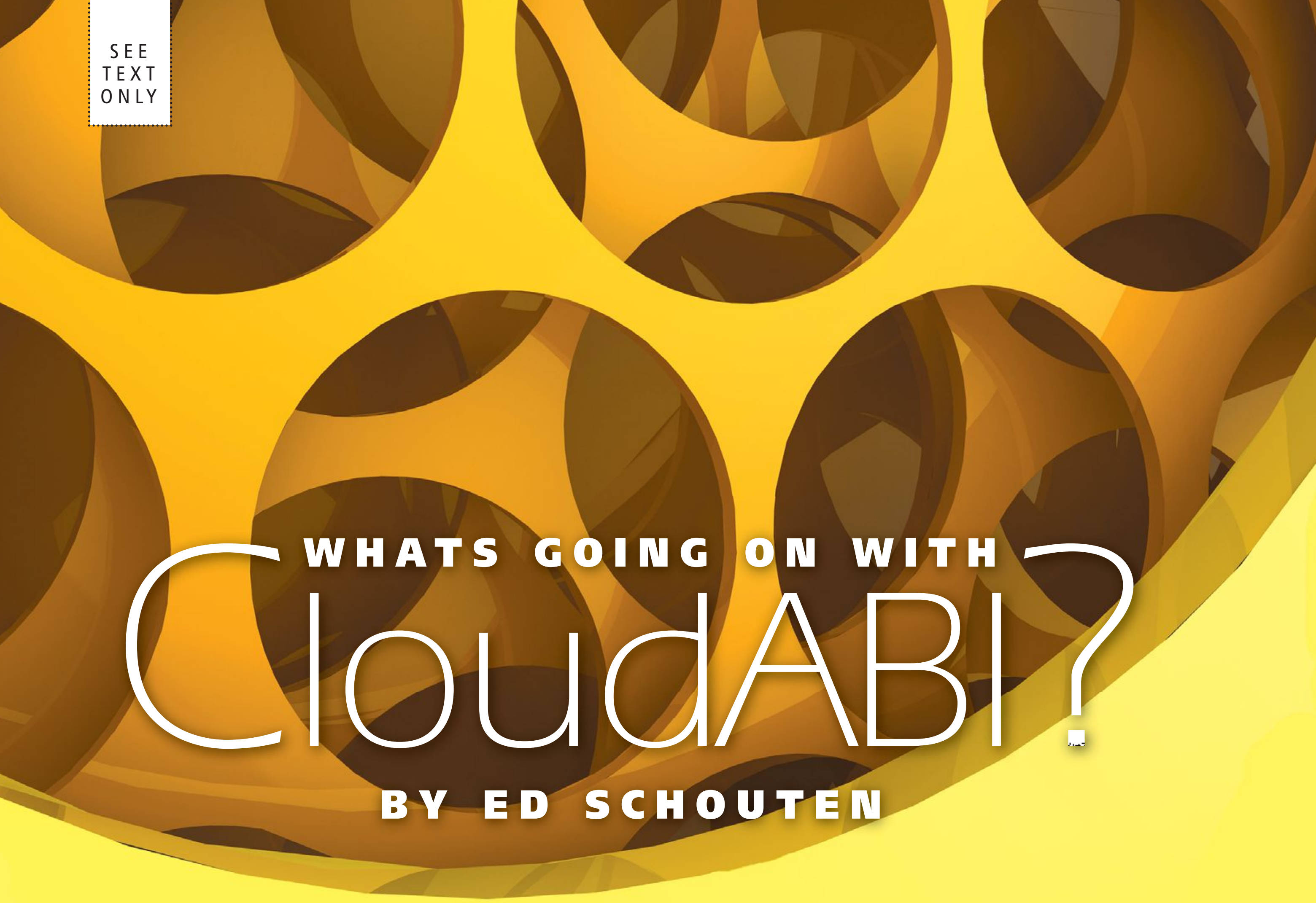
SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

SEE TEXT ONLY



WHATS GOING ON WITH

CloudABI?

BY ED SCHOUTEN

The September/October 2015 issue of *FreeBSD Journal* featured an article on CloudABI, an open-source project I started working on earlier that same year. As a fair amount of time has passed since the article appeared, let's take a look at some of the developments that have taken place in the meantime and what is being planned next. But first, a recap of what CloudABI is and what using it looks like.

CLOUDABI IN THEORY

CloudABI is a runtime for which you can develop POSIX-like software. CloudABI is different from FreeBSD's native runtime in that it enforces programs to use dependency injection. Dependency injection is a technique that is normally used in the field of object-oriented programming to make components (classes) of a program easier to test and reuse. Instead of designing classes that attempt to communicate with the outside world directly, you design them so that communication channels are expressed as other objects on which the class may perform operations (dependencies).

A good example of dependency injection is a web server class that depends on a separate socket object being provided to its constructor. Such a class can be unit tested very easily by passing in a mock socket object that generates a fictive HTTP request and validates the web server's response. This class is also reusable, as support for different network protocols (IPv4 vs. IPv6, TCP vs. SCTP), cryptography (TLS), rate limiting, traffic shaping, etc., can all be added as separate helper classes. The implementation of the web server class itself can remain unchanged. A web server class that directly creates its own network socket through the `socket(2)` system call wouldn't allow for this.

The goal behind CloudABI is to introduce dependency injection at a higher level. Instead of applying it to classes, we want to enforce entire programs to have all of their dependencies injected explicitly on startup. In our case, dependencies are represented by file descriptors. Programs can no longer open files using absolute pathnames. Files are only accessible by injecting a file descriptor corresponding to either the file itself or one of its parent directories. Programs can also no longer bind to arbitrary TCP ports. They can be injected with pre-bound sockets.

What is nice about this model is that it reduces the need for jails, containers, and virtual machines. These technologies are often used to overcome limitations related to the inability to run multiple configurations/versions of the same software alongside, due to a lack of isolation provided by UNIX-like systems by default. With CloudABI, it is possible to obtain such isolation by simply injecting every instance of a program with different sets of files, directories, and sockets.

It becomes very easy for an operating system to sandbox processes following this approach. By requiring all dependencies to be injected, the operating system can simply deny access to everything else. There is no need to maintain security policies separately, as is the case with frameworks like Linux's SELinux and AppArmor. If an attacker manages to take over control of a CloudABI-based process, he/she will effectively only be able to access the resources the process needs to interact with by design.

CloudABI has been influenced a lot by Capsicum, FreeBSD's capability-based security framework. CloudABI differs from Capsicum in that Capsicum still allows you to run startup code without having sandboxing enabled. The process has to switch over to "capabilities mode" manually, using **`cap_enter(2)`**. CloudABI executables are already sandboxed by the time the first instruction of the program gets executed.

The advantage of Capsicum's model is that it makes it possible to integrate sandboxing into conventional UNIX programs, which are typically not designed to have all of their dependencies injected. The advantage of CloudABI's model is that it allows us to remove all APIs that are incompatible with sandboxing. This reduces the effort needed to port software tremendously, as parts that need to be modified to work with sandboxing now trigger compiler errors, as opposed to runtime errors that may be hard to trigger, let alone debug.

A side effect of removing all of these sandboxing-incompatible APIs is that it makes CloudABI so compact that it can be implemented by other operating systems relatively easily. This means that you can use CloudABI to build a single executable that runs on multiple platforms without recompilation.

CLOUDABI IN PRACTICE

To demonstrate what it looks like to use CloudABI in practice, let's take a look at a tiny web server for CloudABI that is capable of returning a fixed HTML response back to the browser.

```
#include <sys/socket.h>
#include <argdata.h>
#include <program.h>
#include <string.h>
#include <unistd.h>

void program_main(const argdata_t *ad) {
   // Extract socket and message from config.
   int sockfd = -1;
   const char *message = "";
   {
      argdata_map_iterator_t it;
      argdata_map_iterate(ad, &it);
      const argdata_t *key, *value;
      while (argdata_map_next(&it, &key, &value)) {
         const char *keystr;
         if (argdata_get_str_c(key, &keystr) != 0)
            continue;
         if (strcmp(keystr, "http_socket") == 0)
            argdata_get_fd(value, &sockfd);
         else if (strcmp(keystr, "html_message") == 0)
            argdata_get_str_c(value, &message);
      }
   }

   // Handle incoming requests.
   // TODO: Actually process HTTP requests.
   // TODO: Use concurrency.
   for (;;) {
      int connfd = accept(sockfd, NULL, NULL);
      dprintf(connfd,
         "HTTP/1.1 200 OK\r\n"
         "Content-Type: text/html\r\n"
         "Content-Length: %zu\r\n\r\n"
         "%s", strlen(message), message);
      close(connfd);
   }
}
```

What you may notice immediately is that CloudABI programs get started through a function called `program_main()`, as opposed to using C's standard `main()` function. The `program_main()` function does away with C's string command-line arguments and replaces it with a YAML/JSON-like tree structure called Argdata. In addition to storing values like booleans, integers, and strings, Argdata can have file descriptors attached to it. This is the mechanism that is used to inject dependencies on startup. This web server expects the Argdata to be a map (dictionary), containing both a socket for accepting incoming requests (`http_socket`) and a HTML response string (`html_message`).

The following shell commands show how this web server can be built and executed. The web server can be compiled using a cross compiler provided by the `devel/cloudabi-toolchain` port. Once built, it can be started with `cloudabi-run`, which is provided by the `sysutils/cloudabi-utils` port. The `cloudabi-run` utility reads a YAML file from `stdin` and converts it to an Argdata tree, which is passed on to `program_main()`. The YAML file may contain tags like `!fd`, `!file`, and `!socket`. These tags are directives for `cloudabi-run` to insert file descriptors at those points in the Argdata tree. Only file descriptors referenced by the Argdata end up in the CloudABI process.

```
$ x86_64-unknown-cloudabi-cc -o webserver webserver.c
$ cat webserver.yaml
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
http_socket: !socket
    type: stream
    bind: 0.0.0.0:8080
html_message: <marquee>Hello, world!</marquee>
$ cloudabi-run webserver < webserver.yaml &
$ curl http://localhost:8080/
<marquee>Hello, world!</marquee>
```

This example shows that CloudABI can be used to build strongly sandboxed applications in an intuitive way. With the configuration passed to `cloudabi-run`, this web server is isolated from the rest of the system completely, with the exception of the HTTP socket on which it may accept incoming connections. By using Argdata, we can also omit a lot of boilerplate code from our web server, like configuration file parsing and socket creation. All of this functionality is implemented by `cloudabi-run` once and can be reused universally.

HARDWARE ARCHITECTURES

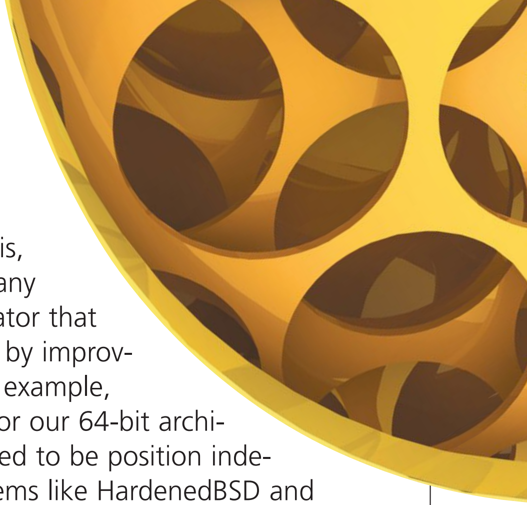
When CloudABI was released in 2015, we only provided support for creating executables for x86-64. As I believe CloudABI is a very useful tool for sandboxing software on embedded systems and appliances as well, we ported CloudABI to also work nicely on ARM64 around the same time the previous article on CloudABI was published. In August 2016, we ported CloudABI to the 32-bit equivalents of these architectures (i686 and ARMv6).

An interesting aspect of porting over CloudABI to these systems was to obtain a usable toolchain. When CloudABI was available only for x86-64, we already used Clang as our C/C++ compiler. Clang is nice in the sense that a single installation can be used to target multiple architectures very easily. It can automatically infer which architecture to use by inspecting `argv[0]` on startup. This meant that we only needed to extend the existing `devel/cloudabi-toolchain` port to install additional symbolic links pointing to Clang for every architecture that we support.

At the same time, we still made use of GNU Binutils to link our executables. Binutils has the disadvantage that an installation can only be used to target a single hardware architecture. Even worse, the Binutils codebase always requires a large number of modifications for every pair of operating system and hardware architecture it should support.

At around the time we started working on supporting more architectures, the LLVM project was making a lot of progress on their own linker, LLD. What is pretty awesome about LLD is that it's essentially free of any operating system specific code. It's capable of generating binaries for many ELF-based operating systems out of the box, simply by using sane defaults that work well across the board. Compared to GNU Binutils, it also has a more favorable license (MIT vs. GPLv3).

When we started experimenting with LLD, we noticed there were still some blockers that prevented us from using it immediately. An important step during the linking process is that the linker



applies relocations: a series of rules stored in object files that describe how machine code needs to be adjusted to point to the correct addresses of variables and functions when being linked into a program or library. We observed that LLD applied several types of relocations incorrectly, causing resulting executables to access invalid memory addresses almost instantly. This was due to the fact that the LLD developers had mainly focused on getting dynamically linked executables to work, whereas CloudABI uses static linkage.

After filing bug reports and sending various patches upstream, we managed to get LLD working reliably for at least x86-64, i686, and ARM64. For full ARMv6 support we had to wait until LLD 4.0 got released, as ARMv6 uses a custom format for C++ exceptions metadata (EHABI) that LLD didn't yet support.

LLD worked so well for us that at one point we decided to stop using GNU Binutils entirely. Together with Google's Fuchsia operating system, CloudABI is now one of the systems that has switched over to LLD completely. The `devel/cloudabi-toolchain` port now installs a toolchain based on LLVM 4.0, setting up symbolic links for `*-unknown-cloudabi-ld` to point to LLD.

OPERATING SYSTEMS AND EMULATORS

One of the original requirements for running CloudABI programs was that you needed an operating system kernel capable of executing them natively. On FreeBSD, this is very easy to achieve, as FreeBSD 11 and later ship with the kernel modules for that by default (called `cloudabi32.ko` and `cloudabi64.ko`, both depending on common code in `cloudabi.ko`). The Linux kernel patchset has also matured over time, but hasn't been upstreamed, meaning that users still need to install custom-built kernels. On systems like macOS, it's undesirable to install a modified operating system kernel.

To lower the barrier for at least experimenting with CloudABI on these systems, we've developed an emulator capable of running CloudABI executables on top of unmodified UNIX-like operating systems. The emulator works by mapping the executable in the same address space and jumping to its entry point. Code is executed natively, without being interpreted or recompiled dynamically. System calls end up calling into the emulator, which forwards them to the host operating system.

While working on this, we wanted to prevent any complexity in the emulator that could easily be avoided by improving CloudABI itself. For example, CloudABI executables for our 64-bit architectures are now required to be position independent. Whereas systems like HardenedBSD and OpenBSD are mainly interested in using Position Independent Executables (PIE) to allow for Address Space Layout Randomization (ASLR), we see it as a useful tool for guaranteeing that CloudABI executables can be mapped by the emulator without conflicting with address ranges used by the emulator internally.

Another improvement we've made is that CloudABI executables no longer attempt to invoke system calls through special hardware instructions like `int 0x80` and `syscall` directly. This is important, as we don't want CloudABI executables to call into the host system's kernel while emulated. They must call into the emulator instead. The runtime is now required to provide an in-memory shared library (a virtual Dynamic Shared Object, vDSO) to CloudABI executables on startup, exposing one function for every system call supported by the runtime. When running in an emulator, the vDSO points to system call handlers in the emulator. When running natively, the kernel provides a vDSO to the process that contains tiny wrappers that do use the special hardware instructions to force a switch to kernel mode:

```
ENTRY(cloudabi_sys_fd_sync)
  mov $15, %eax
  syscall
  ret
END(cloudabi_sys_fd_sync
```

An advantage of using a vDSO this way is that it makes it a lot easier to add and remove system calls over time. As system calls are now identified by strings, not numbers, third parties can easily place extensions under a custom prefix that doesn't clash with CloudABI's set of system calls (e.g., `acmecorp_sys_*`, as opposed to `cloudabi_sys_*`). Programs can easily detect which system calls are present and absent during startup by simply scanning the vDSO's symbol table.

Finally, we've also made some improvements to the way CloudABI implements Thread-Local Storage (TLS). It is now designed in a way that the emulator can more efficiently switch between the context used by the host and guest process. This is achieved by requiring that the Thread Control Block (TCB) of the guest always retains a pointer to the TLS area of the host. When performing a system call, the emulator can temporarily reinstate its own TLS area by extracting it from the TCB of its guest.

One of the goals behind building an emulator using this approach is that having an easy way of embedding the execution of CloudABI programs is useful for many purposes unrelated to emulation. One can now design a system call tracing utility like `truss(8)` entirely in user space without depending on any special kernel interfaces like `ptrace(2)`. Other interesting use cases include user space deadlock detectors for multithreaded code, and fuzzers to inject random failures.

The user space emulator for CloudABI has in the meantime been integrated into `cloudabi-run` and can easily be enabled by passing in the `-e` command line flag. Below is a transcript of how one can build and run a CloudABI program on macOS.

```
$ cat hello.c
#include <argdata.h>
#include <program.h>
#include <stdio.h>
#include <stdlib.h>

void program_main(const argdata_t *ad) {
  int fd = -1;
  argdata_get_fd(ad, &fd);
  dprintf(fd, "Hello, world!\n");
  exit(0);
}
```

```
$ x86_64-unknown-cloudabi-cc -o hello hello.c
$ cat hello.yaml
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
!fd stdout
$ cloudabi-run hello < hello.yaml
Failed to start executable: Exec format error
$ cloudabi-run -e hello < hello.yaml
Hello, world!
```

BETTER C++ SUPPORT

When CloudABI was developed initially, our main focus was to get code written in C to work. After CloudABI's C library became relatively complete and a fair number of packages for software written in C started to appear, we shifted our focus toward improving the experience of porting software written in C++ to CloudABI.

Early on we had managed to get LLVM's C++ runtime libraries (**libcxx**, **libcxxabi**, and **libunwind**) to work, but they still required a lot of local patches. In many cases, these patches were cleanups not specific to CloudABI. They made the code more portable in general. Since the previous article was published, we've been able to get almost all of these patches integrated. At the same time, we've also been able to package Boost, a commonly used framework for C++.

An interesting piece of software written in C++ that we've ported to CloudABI is LevelDB. LevelDB is a library that implements a heavily optimized sorted key-value store, using a data structure called a log-structured merge-tree. It is used within Google as a building block for BigTable, a database system that powers many of their web services.

Porting LevelDB really demonstrated the strength of CloudABI: by omitting any interfaces incompatible with Capsicum, it was trivial for us to find the parts of code that needed to be patched up to work well with sandboxing. In the case of LevelDB, it pointed us straight to **leveldb::Env**, the class that implements all of the filesystem I/O. We've made it possible to use LevelDB in sandboxed software by changing this class to hold a file descriptor of a directory to which filesystem operations should be confined. Whereas you would normally use LevelDB's API to access a database as follows:

```
leveldb::Options opt;
opt.create_if_missing = true;
opt.env = leveldb::Env::Default();
leveldb::DB *db;
leveldb::Status status = leveldb::DB::Open(opt, "/var/...", &db);
db->Put(leveldb::WriteOptions(), "my_key", "my_value");
```

You can make use of LevelDB on CloudABI like this:

```
leveldb::Options opt;
opt.create_if_missing = true;
opt.env = leveldb::Env::DefaultWithDirectory(db_directory_fd);
leveldb::DB *db;
leveldb::Status status = leveldb::DB::Open(opt, ".", &db);
db->Put(leveldb::WriteOptions(), "my_key", "my_value");
```

With libraries like these readily available, we're now able to start working on bringing entire programs to CloudABI. One of the lead developers of Bitcoin, Wladimir van der Laan, happened to discover that most of the libraries used by Bitcoin's reference implementation, like Boost and LevelDB, had already been ported and packaged by us. As a result, Wladimir has been able to successfully port **bitcoind** to CloudABI.

The initial goal of this project is to isolate Bitcoin from other processes running on the same system. A future goal is to use CloudABI to perform privilege separation, so that security flaws in the network protocol handling can't be used by an attacker to obtain direct access to the wallet storing the user's Bitcoins.

RUNNING SANDBOXED PYTHON CODE

In late 2015, I gave a talk about CloudABI at the 32C3 security conference in Hamburg. During this talk, I briefly mentioned that I had plans to port the Python interpreter to CloudABI. This seemed to have made an impression on the audience, as I got an email from Alex Willmer not long after the conference, offering to help out.

During the months that followed, Alex and I worked together a lot, coming up with patches both for Python and CloudABI's C library to get Python to build as cleanly as possible. After getting the interpreter to build, Alex worked on extending Python's module loader, **importlib**, to allow you to include paths to **sys.path** using directory file descriptors, as opposed to using

pathname strings. Finally, I worked on integrating the Python interpreter with Argdata, so that it can be launched through `cloudabi-run`.

Below is a demonstration of what it looks like to run a simple "Hello, world" script using our copy of Python. During its lifetime, the interpreter only has access to Python's modules directory, the script we're trying to execute, and the terminal to which the script should write its message .

```
$ cat hello.py
import io
import sys

stream = io.TextIOWrapper(sys.argdata['terminal'],
     encoding='UTF-8')
print(sys.argdata['message'], file=stream)
$ cat hello.yaml
%TAG ! tag:nuxi.nl,2015:cloudabi/
---
path:
- !file
    path: /usr/local/x86_64-unknown-cloudabi/lib/python3.6
script: !file
    path: hello.py
args:
    terminal: !fd stdout
    message: Hello, world!
$ cloudabi-run \
    /usr/local/x86_64-unknown-cloudabi/bin/python3 < hello.yaml
Hello, world!
```

Though it may at first seem somewhat complex to run Python this way, the requirement for injecting all dependencies of Python explicitly does have the advantage that it's now a lot easier to maintain multiple Python environments. Each of these environments may have different versions of third-party modules installed. This approach effectively means there is no longer any need to use Python's own `virtualenv` to achieve isolation between environments. This can already be accomplished by injecting Python with the right set of file descriptors.

The example given above is, of course, only intended to scratch the surface of how you can use CloudABI's copy of Python. On the CloudABI development blog, you can find an article that explains how you can use `socketserver` and `http.server` to build your own sandboxed web services. In the meantime, we're also working on porting the Django web application framework. A preliminary version that is capable of serving requests has already been packaged.

FORMALIZATION OF THE BINARY INTERFACE

All of CloudABI's low-level data types and constants were originally defined through a set of C header files. The problem with these header files was that they became pretty complex over time. As we allow you to run 32-bit CloudABI executables both on 32-bit and 64-bit systems, we had to maintain copies of the definitions that either assumed the target's native pointer size (used by user space) or that assumed 32-bit or 64-bit pointers explicitly (used by kernel space). CloudABI's system call table was translated to FreeBSD's in-kernel format and kept in sync by hand.

To clean this up, Maurice Bos has worked on a project to formalize CloudABI. All CloudABI data types, constants, and system calls are now described in a programming language independent notation in a file called `cloudabi.txt`. Below is an excerpt of what the definitions related to the `cloudabi_sys_fd_read()` system call look like:

```
opaque uint32 fd
  | A file descriptor number.

struct iovec
  | A region of memory for scatter/gather reads.
  range void buf
     | The address and length of the buffer to be filled.

syscall fd_read
  | Reads from a file descriptor.
  in
     fd           fd
        | The file descriptor from which data should be
        | read.
     crange iovec iovs
        | List of scatter/gather vectors where data
        | should be stored.
  out
     size         nread
        | The number of bytes read.
```

From this file, we now automatically generate C header files, system call tables, vDSOs, and HTML/Markdown documentation. We eventually want to use this framework to generate low-level bindings for other languages as well (e.g., Rust, Go), so that they can be ported to CloudABI without depending on any definitions that are copied by hand.

ARGDATA: NOW A SEPARATE LIBRARY

Though Argdata was originally designed just to be used for passing startup configuration to CloudABI programs, it's actually a pretty flexible and efficient binary serialization library under the hood. Compared to MessagePack, a similar encoding format, it allows for efficient random access of data without performing full deserialization. Compared to `libnv(3)`, a serialization library in FreeBSD's base system, it has a more conventional data model and a simpler API.

Earlier this year, Maurice Bos expressed interest in using Argdata as a format for serializing RPC messages in a non-CloudABI application written in C++. As the Argdata library was tightly integrated into CloudABI's C library, Maurice spent some time making it more portable and turning it into a separate library.

At the same time, he also wrote some really good C++ bindings for Argdata, making use of various features provided by modern revisions of the language (C++11, C++14, C++17). Maps and sequences can be iterated using C++'s range-based for loops. By making use of `std::optional<T>`, C++'s implementation of a '"maybe type," it becomes easier to deal with potential type mismatches. Strings are returned as `std::string_view` objects, meaning they can be used by C++ code without copying them out of the serialized data or allocating copies on the heap.

Below is an example of what the C++ bindings look like when used in practice. When compared to the web server provided in the introduction that uses the C API, the C++ code is a lot more compact and easier to understand.

```
#include <argdata.hpp>
#include <cstdlib>
#include <optional>
#include <program.h>

void program_main(const argdata_t *ad) {
   // Scan through all configuration options and
   // extract values.
   std::optional<int> database_directory, logfile, http_socket;
   for (auto [key, value] : ad->as_map()) {
      if (auto keystr = key->get_str(); keystr) {
         if (*keystr == "database_directory")
            database_directory = value->get_fd();
         else if (*keystr == "logfile")
            logfile = value->get_fd();
         else if (*keystr == "http_socket")
            http_socket = value->get_fd();
      }
   }

   // Terminate if we didn't get started with all
   // necessary descriptors.
   if (!database_directory || !logfile || !http_socket)
      std::exit(1);

   …
}
```

NEXT PROJECT: CLOUDABI FOR KUBERNETES?

Over the last couple years there has been a lot of development in the area of cluster management systems. These systems allow you to treat a large group of servers as a single pool of computing resources on which you can schedule jobs. One system that is gaining popularity is Kubernetes, designed by Google and funded through the Linux Foundation's recently founded Cloud Native Computing Foundation (CNCF). Kubernetes can run any software that has been packaged as a container (using Docker, rkt, etc.).

What I've observed while using Kubernetes is that it has a number of quirks stemming from the fact that it has to be able to run software that is not dependency injected. For example, because programs running in containers are free to bind to arbitrary network ports, every job ("pod") in the cluster must have a unique IPv4 address. This makes Kubernetes consume a lot of network addresses and makes routing tables of nodes in the cluster very complex. Conversely, as most conventional software can only connect to a single network address to reach backend services, Kubernetes does lots of TCP-level load balancing for internal traffic, which makes tracing and debugging very complicated.

The fact that jobs in the cluster are able to create arbitrary network connections also means that security between pods can only be achieved if all containers running in the cluster are configured to make use of cryptography, authentication, and authorization (e.g., using SSL with a per-service custom trust chain). Unfortunately, people hardly ever bother setting that up correctly, meaning it is generally not safe to operate a single Kubernetes cluster for a multitenant environment.

An interesting development for us is that as of version 1.5, Kubernetes no longer communicates with the system's container engine directly. When Kubernetes wants to start a container on a node in the cluster, it sends an RPC for that to an additional process, called the Container Runtime Interface (CRI). This mechanism is intended to allow people to experiment with custom container formats more easily by developing their own CRIs.

In our case, we could implement our own CRI that allows us to run CloudABI processes directly on top of Kubernetes. By using CloudABI, we can enforce jobs running on the cluster to have all of their network connectivity injected by a helper process. This helper process can ensure all traffic in the cluster is encrypted, authorized, and load balanced. Software running on the cluster will no longer need to care about the model of the underlying network.

WRAPPING UP

I hope this article has shown that a lot of interesting things have happened with CloudABI over the last year and that there are even more exciting things planned. As CloudABI is part of FreeBSD 11 and most new features have already been merged into 11-STABLE, CloudABI has become an easy-to-use tool for creating secure and testable software. If you maintain a piece of software that could benefit from this, be sure to experiment with building it for CloudABI.

As most of the discussion around CloudABI takes place on IRC, feel free to join #cloudabi on EFnet if you have any questions or simply want to stay informed about what's going on.

LINKS

CloudABI on FreeBSD: https://nuxi.nl/cloudabi/freebsd/

CloudABI development blog: https://nuxi.nl/blog/

CloudABI on GitHub: https://github.com/NuxiNL

BitCoin for CloudABI: https://laanwj.github.io/

ED SCHOUTEN has been a developer at the FreeBSD Project since 2008. His contributions include FreeBSD 8's SMP-safe TTY layer, the initial import of Clang into FreeBSD 9, and the initial version of the vt(4) console driver that eventually made its way into FreeBSD 10.

CloudABI has been developed by the author's company, Nuxi, based in the Netherlands. It will always be available as open-source software, free of charge. Nuxi offers commercial support, consulting, and training for CloudABI. If you are interested in using CloudABI in any of your products, be sure to get in touch with Nuxi at info@nuxi.nl.

DELL
BUILD SOMETHING GREATER.
Dell is now part of the Dell Technologies family. So you can make your mark in everything from storage and big data to the Internet of Things.
Apply at Dell.com/careers

SEE TEXT ONLY

new faces of FreeBSD

BY DRU LAVIGNE

This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community. This month, the spotlight is on Eugene Grosbein, a longtime contributor who transitioned to a ports committer in March.

Tell us a bit about yourself, your background, and your interests.

● I was born in and live in Novokuznetsk (founded in 1618), West Siberia, Russian Federation. I lived in Novosibirsk for several years while studying at Novosibirsk State University (NSU). I have been interested in mathematics since secondary school when I got my first programmable calculator in the late '80s. That was the Russian model MK-61 with 15 addressable memory registers and 4 operating registers combined as a stack for reverse Polish notation operators (https://upload.wikimedia.org/wikipedia/ru/2/27/Elektronika-Mk-61.JPG). It had memory for 105 program instructions. My first programs were created using its machine code.

Later I took a computer class that had Z80A-based MSX-2 computers and then an IBM PS/2 pilot class (80286-based Model 30). Writing simple video games was as much fun for me as playing them. I understood that computers would be my future fun and livelihood.

I've enjoyed reading lots of books. My father was a literature teacher and our home library was large. Also, I studied music and played guitar in a school band. I must admit that computer studies banished playing music from my life, but my taste for good music persisted.

"For those interested in becoming a FreeBSD committer: you can do it! Help us improve FreeBSD code, docs, or ports. Improve yourself while you have fun doing this, talk to other contributors and committers, offer help and ask for help from them, and you'll make it. We need new people here!." —EUGENE GROSBEIN

How did you first learn about FreeBSD and what about FreeBSD interested you?

● I started to use FreeBSD 2.2.5-RELEASE in 1996 during my early university days, as a beginner C and Java programmer. FreeBSD was already used on some computers in the Institute of Computational Mathematics and Mathematical Geophysics, also known as the Computing Center of the Russian Academy of Sciences, Siberian Branch, where I was working and doing my diploma work. Those were the good old times when FreeBSD had /etc/sysconfig instead of /etc/rc.conf and one could use FreeBSD with 4 megabytes of RAM, but sysinstall needed 5 megabytes or early-activated swap.

Initially, I was only familiar with Microsoft operating systems: MSX-DOS, MS-DOS, and early versions of Windows. FreeBSD opened a whole new universe for me—the open-source approach and the Unix world as well as the ability to change systems at once, network transparent graphics, and lots of other things.

I continued to use FreeBSD after I graduated from NSU with a degree in mathematics in

1999, when I installed 3.2-RELEASE on my home desktop and started to use FreeBSD at work in an ISP environment (2.2.8-RELEASE). That was when I started to contribute to the FreeBSD Project by filing Problem Reports using GNATS (PR 15301 was my first one).

How did you end up becoming a committer?

● In 2001, I created my first port, databases/libudbc (retired now), for Openlink Universal DataBase Connectivity SDK, which I used to connect a FreeBSD-based web server to a DEC RDB database running under VAX/OpenVMS. Since then I have been on the FreeBSD "Additional Contributors" list. In the meantime, I continued to employ FreeBSD in several ISP and other commercial environments. I even ran 64-bit FreeBSD 4.x/alpha for a time. Since then, I've filed about 340 PRs, submitting patches when I could. I think the number of my unclosed reports just reached critical mass so Vsevolod Stakhov and Andrej Zverev offered me a promotion to ports commit bit and became my mentors. We talked about it on an IRC channel and I gratefully agreed.

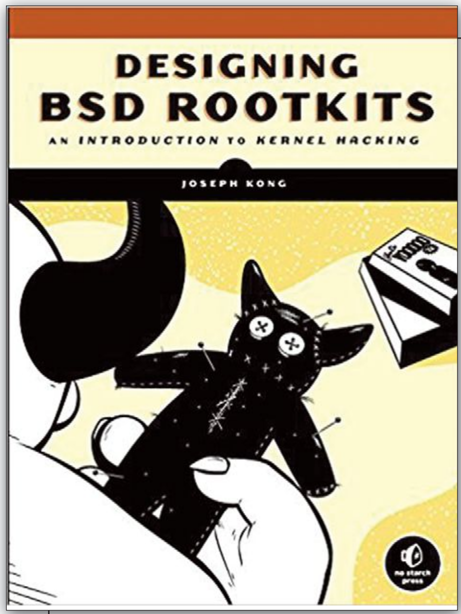
How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

● In spite of my history with the FreeBSD Project as a user and contributor, being a committer proved to be a different matter. There are a lot of things to be learned from the Committers Guide in regard to respecting other people's work and keeping the repository contents maintainable and consistent. But, it's lots of fun to see how you can make it better. ●

DRU LAVIGNE is a doc committer for the FreeBSD Project and Chair of the BSD Certification Group.

SEE
TEXT
ONLY

BOOKreview by Steven Kreuzer



Designing BSD Rootkits:
An Introduction to Kernel Hacking
by Joseph Kong

Publisher	*No Starch Press (2007)*
Print List Price	*$29.95*
Digital List Price	*$23.95*
ISBN-10	*1593271425*
ISBN-13	*978-1593271428*
Pages	*142 pages*

It's hard to believe that it was 10 years ago when I stumbled across *Designing BSD Rootkits*, by Joseph Kong, as I was wandering around a local bookstore. Not only was this a book on a technical topic that I happen to find pretty fascinating, but it also used FreeBSD as the reference platform! Since the focus of this issue is security, I thought it would be fun to dust off my copy and take another look.

At this point you might be asking what exactly is a rootkit? Once an attacker gains unauthorized access to your system, the goal is usually to avoid detection and maintain control of the machine for as long as possible. This is usually accomplished by deploying a rootkit, which is a collection of utilities designed to hide the intrusion as well as to maintain privileged access. While this book focuses on the FreeBSD operating system, the concepts that are covered can be applied to other operating systems as well. Even if you do not work with FreeBSD on a day-to-day basis, you will still gain valuable insights into computer security that can be applied to pretty much any system.

One of the things I love about this book is that it cuts right to the chase. This first chapter immediately dives into the topic of loadable kernel modules, which is the double-edge sword that allows your system administrator to add functionality to your running system without having to reboot, but is used by an attacker to load malicious code that can help cover their tracks. Kong wastes no time, because on the fourth page of this book you are presented with a bare bones kernel module which you can build, load, and unload on your machine. Following a tutorial style format, each chapter slowly builds upon the previous one, introducing more advance functionality. By the end of the book your simple kernel module which just printed 'Hello, world!' will be a complete and functional rootkit capable of evading host-based intrusion detection systems such as tripwire.

In the final chapter, the focus shifts from offense to defense with a chapter dedicated to the detection of rootkits. Amusingly, this is also the second shortest chapter with very few pieces of example code. The overall theme is that once a rootkit is in place, detection isn't easy. Software designed to detect a rootkit can just as easily be subverted by the very rootkit it is looking for, and the kernel is a very big place that offers lots of places to hide. However, since you are now armed with a much better understanding of what is actually going on under the hood, you will be in a superior position should you ever have to do battle with a rootkit.

While the book is thin, weighing in at only 142 pages, it manages to pack in a wealth of information and example code. To get the most out of this book, it will be helpful to have some understanding of both C and assembly as well as some knowledge of FreeBSD kernel internals. However, I found Kong's tutorial-style approach combined with the very well documented code examples made the concepts incredibly accessible. The fact that each chapter provided at least one real-world application for each topic that he covers makes it even easier to experiment and build upon the examples on your own. You could read the book from cover to cover in a few hours, but I would highly recommend that you sit yourself down in front of your computer with a copy of this by your side while you play around with the provided examples. It becomes quite easy to quickly lose a few hours as you load and unload a kernel module designed for nefarious purposes.

Don't let the fact that the book was published a decade ago scare you away. Not only does this book serve as a great introduction to FreeBSD kernel module programming, the methods and tactics that are covered are still in use today. One thing to note is that a lot has changed in FreeBSD, and when this book was first written, the author was using FreeBSD 6.0-STABLE on a 32-bit platform. Not all the code examples may properly compile on a newer system, so it becomes an exercise for the reader to debug and update the provided examples. With all that said, I highly recommend you add a copy to your bookshelf. ●

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, two daughters, and dog.

SEE TEXT ONLY

svn UPDATE

by Steven Kreuzer

Like everything else in life, security is a trade-off, and it usually comes at the expense of the end user losing functionality or having to endure a decrease in performance or throughput. What this usually means is that unless you have people and/or an organization that have decided to take security seriously, some of the first things which get turned off are the firewalls or the access control security policies. To make matters even worse, sometimes the security will cause your application to break in a nonobvious way, and taking the sledgehammer approach of turning everything off until it works again is all too common. At the other end of the spectrum, you can be left with a false sense of security and have machines that are actually vulnerable because of a very subtle typo you made in a complex security subsystem with difficult-to-understand syntax.

Security is hard, and it's even harder to get it right. One of the great things about the FreeBSD Project is that system security has always been a major focus. Developers spend a lot of time and effort to provide a secure and reliable system so that someone with little to no experience using FreeBSD can build a new machine and connect it directly to the Internet and be reasonably confident that nothing bad will happen. Over the past few years, FreeBSD has been used as a proving ground for some new and innovative concepts in computer security that are both lightweight and mostly transparent, resulting in a much better experience for both system administrators and end users alike. Since the topic of this issue is security, I wanted to use this installment of svn update to highlight some of the recent improvements designed to keep you and your data safe, be it improvements in encryption, extending FreeBSD to support hardware security features that are being baked into the CPU itself, or just the ongoing work to explicitly define what an application can and cannot do with Capsicum.

Add ipfw_pmod kernel module (https://svnweb.freebsd.org/changeset/base/316435)

This new module is designed for modification of packets from any protocol, only implementing TCP MSS modification at this time. It adds the external action handler for "tcp-setmss" action. A rule with tcp-setmss action performs an additional check for protocol and TCP flags. If SYN flag is present, it parses TCP options and modifies the MSS option if its value is greater than the configured value in the rule.

Capsicumize pom (https://svnweb.freebsd.org/changeset/base/317165)

I had a good laugh when I saw this commit because I was not aware that out-of-the-box FreeBSD provided the ability to report the phases of the moon. While this appears to be a fairly trivial application with little attack surface, it has been converted to run in a capsicum sandbox so you can rest easy at night.

Set the arm64 Execute-never bits in more places (https://svnweb.freebsd.org/changeset/base/316761)

Each memory region on arm64 can be tagged as not containing executable code. If the Execute-never, XN, bit of the descriptor is set to 1, any attempt to execute an instruction in that region results in a permission fault. Set the Execute-never bits when mapping device memory, as the hardware may perform speculative instruction fetches. Set the Privileged Execute-never bit on userspace memory to stop the kernel if it is tricked into executing it.

Enable the process state bit to disable access to userspace from the kernel on ARMv8.1 (https://svnweb.freebsd.org/changeset/base/316756)

ARMv8.1 introduced a new Privileged Access-never (PAN) state bit. This bit provides control that prevents privileged access to user data unless explicitly enabled, which provides additional security against possible software attacks.

3BSD-licensed implementation of the ChaCha20 stream cipher, intended or used by the upcoming arc4random replacement (https://svnweb.freebsd.org/changeset/base/316982)

The ChaCha20 cipher is a high-speed cipher published by Daniel J. Bernstein in 2008. It is considerably faster than AES in software-only implementations due to its use of CPU-friendly Addition-rotation-XOR operations.

Replace the RC4 algorithm for generating in-kernel secure random numbers with ChaCha20 (https://svnweb.freebsd.org/changeset/base/317015)

Writing software is hard but we are pretty fortunate that contributions being made to FreeBSD are coming from a pool of very talented people. However, these developers are human, and humans sometimes make mistakes. But we have a very active community of people who continually audit the code base looking for potential security vulnerabilities, both large and small. While not as exciting as some of the other changes I mentioned, I thought it would be interesting to take a look at a few small snippets of code that appear innocuous, but could have potentially been used as an avenue to compromise your system.

Use strlcpy to appease static checkers in dummynet (https://svnweb.freebsd.org/changeset/base/316777)

Prevent some heap overflows in restore(8) (https://svnweb.freebsd.org/changeset/base/316799)

Prevent possible sscanf() overflow in mixer(8) (https://svnweb.freebsd.org/changeset/base/317596)

Fix some trivial argv buffer overruns in ctm(1) (https://svnweb.freebsd.org/changeset/base/316795)

Avoid possible overflow via environment variable in loader(8) (https://svnweb.freebsd.org/changeset/base/316771)

Fix an out-of-bounds write when a zero-length buffer is passed (https://svnweb.freebsd.org/changeset/base/316768)

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, two daughters, and dog.

SUMMER OF 2017 BY DRU LAVIGNE

2017

Events Calendar

The following BSD-related conferences will take place during the summer of 2017.

Linuxhotel Hackathon and DevSummit • July 7–9 • Essen, Germany

https://wiki.freebsd.org/DevSummit/201707 • The 2nd annual FreeBSD Hackathon and DevSummit will be held in the park-like setting of the Linuxhotel Villa Vogelsang. Registration is required and there is a nominal fee to cover lodging and food.

BSDCam • August 2–4 • Cambridge, UK

https://wiki.freebsd.org/DevSummit/201708 • The Annual Cambridge FreeBSD DevSummit will take place at the University of Cambridge. Registration is required, lodging is available, and there is a nominal fee to cover meals.

Openhelp • August 12–16 • Cincinnati, OH

https://conf.openhelp.cc/ • This annual conference brings together leaders in open-source documentation and support, as well as people from across the technical communications industry who are interested in community-based help. Several members of the FreeBSD documentation team will be in attendance and there will be a FreeBSD documentation sprint on August 14 and 15 (in-person and on #bsddocs on IRC EFNET). There is a nominal registration fee to attend this event.